INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

NOTE TO USERS

This reproduction is the best copy available

UMI

Performance Analysis Based on Operational Components in An Object Oriented Framework

by

Bingjun Li

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Information and Systems Science

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

November 1998

©1998, Bingjun Li

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36941-2

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**Performance Analysis Based on Operational Components in
An Object Oriented Framework**

submitted by Bingjun Li,
in partial fulfillment of the requirements for the degree of Master of Science

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

December 25, 1998

Abstract

The thesis integrates the use case composition in OO design with resource demand aggregation for performance analysis. New design concepts (operational component, abstract MSC, and etc.) have been proposed for this integration. Extensions to message sequence charts are proposed to include performance information which is necessary for the performance analysis of object oriented design. Rules for use case composition for O-O design are defined. Structural graph reduction rules are adopted for the creation of operational components using composition approach in object oriented design.

A detailed case study for use case composition of application protocol design with a known framework is conducted. Both decomposition and composition approaches have been used in the design. Top-down decomposition is used for use case driven O-O design for system architecture design, whereas composition is used for operational component design and its performance analysis. The case study using operational component concept has suggested that operational components can describe performance correctly for a framework while traditional MSC can not; mapping from use cases to operational components is straightforward. These results shows that using (concrete and abstract) operational component helps to fill the gap between use case design and object orientation while performance information in operational components can be used for performance analysis. Performance measurement was done using Jprobe measurement tool.

to my family....

Acknowledgments

The thesis is the result of a comprehensive study under the supervision of my thesis supervisor, Prof. Dorina Petriu. Dr. Petriu's unflagging enthusiasm, continuous moral and practical support, persistent optimism, and wide practical experience in both object oriented design and performance analysis have provided very strong influence on the author to continue the study in the direction of practical application.

Mr. Mahad Ahmed helped to get the Jacob2 framework package from Prof. Pekka Nikander, Faculty of Information Technology, Helsinki University of Technology. Tero Hasu at Telecommunication Software and Multimedia Laboratory of Helsinki University of Technology told me useful information about compile settings used for the Jacob framework.

I thank my advisor Prof. J.A. Rolia for his helps when I selected my graduate courses. I thank the Professors at the Department of Systems and Computer Engineering and the School of Computer Science at Carleton University, and at the Department of Electrical Engineering and the Department of Mathematics and Statistics at the University of Ottawa for their great teachings.

Finally none of this work would have been possible without the support of the author's family.

Table of Contents

<i>Abstract</i>	<i>iii</i>
<i>Acknowledgments</i>	<i>iv</i>
<i>Table of Contents</i>	<i>v</i>
<i>List of Figures</i>	<i>x</i>
<i>List of Tables</i>	<i>xiv</i>
Chapter 1 Introduction	1
Chapter 2 Literature Review And Analysis	7
2.1 Composition in Object Oriented Design.....	5
2.2 Message Sequence Chart	7
2.3 Performance Analysis Algorithms	10
2.4 Network Protocol Design.....	18
2.5 Summary	32
Chapter 3 Use Case Composition Rules	34

3.1	Concepts.....	34
3.1.1	Extension to Component and Interface Concepts.....	34
3.1.2	Proposed Basic Structures.....	36
3.1.3	Extension Required for MSC and Performance Analysis Algorithms	41
3.2	Extension of Message Sequence Chart	42
3.2.1	Message Sequence ID	42
3.2.2	Add Performance Information for Alternation and Loop in MSC	46
3.2.3	Composition of Message Sequence Chart in a Design ..	48
3.2.4	Abstract MSC and Abstract Operational Component....	49
3.3	Performance Analysis Algorithms for Use Case Composition	52
3.3.1	Rules for Sequential MSC	53
3.3.2	Rules for Loop in MSC.....	58
3.3.3	Rules for Alternation in MSC	59
3.3.4	Rules for AMSC	60
3.4	Summary	64

Chapter 4 Case Study: Implementation of An Application Protocol

Using Java Conduit Framework.....	65
4.1 System and Architecture Design.....	65
4.1.1 Use Cases	65
4.1.2 Analysis and Design	68
4.1.2.1 Mapping from Use Case to Operational Components	68
4.1.2.2 Mapping from Operational Components to Classes ..	71
4.2 Design of Operational Components: Sessions.....	77
4.2.1 CodeSession.....	82
4.2.2 StructSession.....	89
4.2.3 Discussions about Session	93
4.3 Design of Operational Components: Protocols.....	94
4.3.1 Concrete Operational Components	95
4.3.2 MSC in the Exact Java Implementation without	
Operational Component Concept.....	101
4.4 Summary and Discussion.....	104
Chapter 5. Performance Measurement.....	106

5.1	Introduction.....	106
5.2	Measurement Arrangement.....	106
5.2.1	Problems to Use Jprobe for Framework Measurement	106
5.2.2	Arrangement for Framework Measurement and Test Cases	107
5.3	Measurement Results.....	113
5.3.1	CompositeCodeSessionFromNet.....	113
5.3.2	CompositeStructSessionFromNet.....	115
5.3.3	CompositeAppProtocolFromNet.....	117
5.3.4	CompositeStructSessionToNet.....	119
5.3.5	CompositeCodeSessionToNet.....	121
5.3.6	CompositeAppProtocolToNet.....	122
5.3.7	Summary of the Performance Data for Each Concrete Operational Component.....	125
5.3.8	Summary of the Performance Data for Each Abstract Operational Component.....	128
5.4	Summary.....	130
Chapter 6. Conclusions.....		131

Appendix 1: Setup of AppProtocol.....	135
--	------------

References	137
-------------------------	------------

List of Figures

Figure 2.1	Message Sequence Chart Ordering	12
Figure 2.2	Activity Diagram	16
Figure 2.3	Sequential Path Reduction Rule	19
Figure 2.4	Repetition-Path Reduction Rule	19
Figure 2.5	Condition-Path Reduction Rule	20
Figure 2.6	Class Diagram for Conduit (summarized from Zweig and Johnson, 1990)	24
Figure 2.7	Double Dispatching Pattern in Conduit (summarized from Zweig, 1991)	25
Figure 2.8	Visitor Pattern: MsgTransporter	27
Figure 2.9	Class Hierarchy and Relationship of Jacob2 Framework	29
Figure 3.1	Structure Definition of Basic Concepts	38
Figure 3.2	Collaboration Diagrams for Gate Connection	39
Figure 3.3	Message Sequence Chart Using Hierarchy Numbering Structure	44
Figure 3.4	Composition of MySubsystem with Hierarchy Numbering Scheme	45
Figure 3.5	Message Sequence Chart Using Simple Numbering	45

Figure 3.6	Composition of MySubsystem with Simple Numbering Scheme	46
Figure 3.7	Performance Information in Alternation	47
Figure 3.8	Loop with n Repetitions	48
Figure 3.9	A Simple Abstract MSC	50
Figure 3.10	A Simple Concrete MSC	51
Figure 3.11	Simple Operational Component Composition Rule	54
Figure 3.12	Composition Rule: Operational Component within Operational Component	57
Figure 3.13	Loop Operational Component Composition Rule	58
Figure 3.14	Alternation Operational Component Composition Rule	59
Figure 3.15	Class Hierarchy of An AMSC Example	61
Figure 3.16	AMSC for AMSCSubA	62
Figure 4.1	High Level use case for an application	66
Figure 4.2	Decomposed “Send Message” use case	67
Figure 4.3	Decomposed “Receive Message” use case	67
Figure 4.4	Combined use cases	69
Figure 4.5	Complete Picture of Message Flowing in Two Directions	70
Figure 4.6	Operational Components related to StructSession	72
Figure 4.7	Operational Components related to CodeSession	73
Figure 4.8	Operational Components related to AppProtocol	74

Figure 4.9	AppProtocol Conduit	75
Figure 4.10	Class Hierarchy and Relationship of AppConduit	76
Figure 4.11	The highest level of AMSC for Session based on Jacob2	77
Figure 4.12	Composed Abstract Operational Component CompositeSession	78
Figure 4.13	The AMSC for CodeSession	82
Figure 4.14	Composed Abstract Operational Component CompositeCodeSession	83
Figure 4.15	CodeSession to network	84
Figure 4.16	Composed Concrete Operational Component CompositeCodeSessionToNet	84
Figure 4.17	CodeSession from network	88
Figure 4.18	Composed Concrete Operational Component CompositeCodeSessionFromNet	88
Figure 4.19	The AMSC for StructSession	89
Figure 4.20	Composed Abstract Operational Component CompositeStructSession	90
Figure 4.21	StructSession to network	91
Figure 4.22	Composed Concrete Operational Component CompositeStructSessionToNet	91
Figure 4.23	StructSession from network	92

Figure 4.24	Composed Concrete Operational Component	
	CompositeStructSessionFromNet	93
Figure 4.25	Run in MessageTransporter	95
Figure 4.26	CompositeAppProtocolFromNet	96
Figure 4.27	CompositeAppPtotocolToNet	97
Figure 4.28	Composed Operational Components	99
Figure 4.29	Information from network arrives at side B of AppProtocol	102
Figure 4.30	Information from network arrives at side A of AppProtocol	103
Figure 5.1	Exact Execution Path between accept and at in	
	CompositeCodeSession	108
Figure A1.1	MSC: Setup of AppProtocol	135

List of Tables

Table 5.1	Performance Measurement for	
	CompositeCodeSessionFromNet (unit: millisecond)	114
Table 5.2	Performance Measurement for	
	CompositeStructSessionFromNet (unit: millisecond)	116
Table 5.3	Performance Measurement for	
	CompositeAppProtocolFromNet (unit: millisecond)	118
Table 5.4	Performance Measurement for	
	CompositeStructSessionToNet (unit: millisecond)	120
Table 5.5	Performance Measurement for	
	CompositeCodeSessionToNet (unit: millisecond)	121
Table 5.6	Performance Measurement for	
	CompositeAppProtocolToNet (unit: millisecond)	124

Chapter 1 Introduction

As more and more software designers use object oriented design for real-time systems for high reusability, the performance analysis for object oriented design becomes more and more important especially for systems with strong performance requirements (such as the network protocol stacks). But object-oriented methods typically defer consideration of performance issues until detailed design or implementation (Rumbaugh, et al., 1991; Booch, 1994). There is no attempt from the part of OO methodologists to integrate performance engineering into the development process of object oriented design.

In the late 80s, Smith introduced the concept of Software Performance Engineering (SPE), which promotes start early and continue throughout all stages of the software development process in order to build performance into the products instead of trying to fix the performance at the end, when the product is built and all the changes are very expensive (Smith, 1990). The concept of Software Performance Engineering (SPE) as a systematic way to use quantitative methods to assess requirements, design and implementation alternatives for software systems was introduced by Connie Smith in the 80s. SPE should start early in the life-cycle, while a wide range of options exists, and continue throughout the whole life-cycle. The purpose is to design and build systems that meet their performance objectives and to eliminate the need for late-fixing of performance problems. SPE is applied in two big steps. Firstly, software execution models (a kind of control-flow models) are built for a few typical, frequently executed

workload scenarios. The execution models specify the activities to be executed in response to predefined events. Best-and-worst case hardware resource requirements for each activity are estimated, and then total average requirements per resource for each workload scenario are obtained. Secondly, the workload description obtained from the software execution model is mapped onto a system execution model (typically a QN model) representing the hardware resources for which the software modules are competing. The system model solution produces performance metrics for device residence times (including queueing delays), device utilization, overall response time, throughput, etc. These values can be mapped back into the software domain, finding the elapsed time, resource utilization, etc., for each software module.

SPE was mostly applied to software systems using a more traditional design approach based on functional decomposition. It is natural to build flow-like software execution models, as required by SPE, in the context of a traditional software development, where the design starts with specifying the sequence of activities, then allocates the activities to various software modules. Structural graph theory and execution model have been well documented in the literature for performance analysis of structured design by Smith (Smith, 1990). An object-oriented system is more complicated because of its intrinsic features such as encapsulation and dynamic/late binding. The encapsulation makes it hard to understand the collaboration among many different objects in the software, whereas the polymorphism makes the execution path hard to trace. The thesis concentrates on resource demand aggregation for OO designs.

Composition is an important approach for object oriented design especially for component oriented design. Is it possible to apply the use case composition to performance analysis? If yes, how and what extensions to the current use case composition methodology are required? Also what extensions to the current performance analysis algorithms are required? The thesis attempts to answer these questions.

During the research, we came across a paper of Smith et al. (Smith, et al., 1997) which manually translated message sequence charts into SPE execution graph (which are more natural to use for structured design). The paper suggested that “Future research is aimed at providing a smooth translation between CASE tools for OOD and SPE evaluation tools”. Although there is more work on translating OO designs to execution graphs (Williams, and Smith, 1998), there is little progress in the literature addressing a performance analysis methodology tailored OO design. The thesis attempts to find a solution to this problem by studying use-case composition and by introducing the concept of operational component, which simplifies the task of performance analysis of OO analysis.

1.1 Objectives of the Study

Use cases are essential to describe the runtime behaviour in object oriented designs. Use case composition is an appropriate approach to easily view a big system at different levels of abstraction. On the other hand, recursive Graph Reduction Rules have been available for the structural design (Smith, 1990). The first objective of the thesis is to find out if it is possible to combine the characteristics of use cases and graph reduction to provide an appropriate method for the performance analysis of object oriented design.

To achieve this purpose, Message Sequence Charts (MSC) are chosen as the presentation of use cases in the thesis [ITU-T, 1996]. Some essential concepts about runtime components are developed from the composition of message sequence charts. Message sequence charts have been widely used for design of OO and non-OO systems. The second objective of the thesis is to propose extensions to current message sequence chart to present necessary information for performance analysis.

The third objective is to apply the current graph reduction rules to operational components with some modifications/extensions.

The use case composition approach helps to solve the difficulty arising from encapsulation for performance analysis. How to address the polymorphism behaviour (dynamic/late binding) of object oriented designs in use case composition is an important aspect of the development for the above three objectives.

The fourth objective is to apply the results from the above three objectives to a detailed case study to illustrate the proposed methodology.

1.2 Contribution of the Study

The study describes concepts about use case composition, proposes extensions to message sequence charts, introduces graph reduction rules to use case composition, and presents a case study with measurement results.

(1) Concepts for use case composition: propose *operational component*, *abstract operational component*, and *abstract message sequence chart* concepts by using pre-existing gate and message concepts with minor modifications.

(2) Add information necessary for performance analysis in the composition of message sequence charts consisting of the sequence numbers and gate information (operational component, interface, etc.). These extensions will make it possible for a CASE tool to conduct the proposed use case composition approach for resource demand aggregation which is necessary in the performance analysis process.

(3) With the extensions to message sequence chart, it is now possible to import the existent graph reduction rules into the use case composition for object oriented design.

A case study is done for the implementation of an application protocol using a known framework for network protocols, named Java Conduit Framework or Jacob [Nikander,

Parssinen, Sahlin, Hoglund, 1997]. The operational component concept and the extensions are used in the design for performance analysis.

Measurements are done to show some real-time results and to illustrate the proposed concepts.

1.3 Outline of the Thesis

In Chapter 2, a literature review and critical analysis for most issues related to the use case composition for performance analysis in object oriented design is presented.

In Chapter 3, the essential concepts for use case composition in object oriented design, the extensions to message sequence chart, and import of graph reduction rules to use case composition with modifications are presented.

In Chapter 4, an application protocol is implemented using Java Conduit Framework [Nikander, Parssinen, Sahlin, and Hoglund, 1997; Parssinen, 1997]. In the design, the operational component concept is used for both use case composition and performance analysis. Coding is done using VisualAge Java on Windows NT.

In Chapter 5, each concrete operational component from the previous chapter is designed as a test case. Performance measurements are done for the application protocol implemented in Chapter 4 on Windows NT.

Chapter 6 provides a summary of conclusions. Suggested activities for further research in this field are also proposed.

Chapter 2 Literature Review And Analysis

It is important to have a complete picture about what has previously been presented in the literature in order to understand the context of the thesis research and to address the needs for future research.

In this chapter, the literature review and critical analysis are done on composition in object oriented design, message sequence chart, performance analysis algorithms, and network protocol design. The direction of the thesis research is also described.

2.1 Composition in Object Oriented Design

Use of Composition In Object Oriented Design

A lot of industrial O-O systems are build up on some reliable class/component libraries. Reusable software components can be created based on the composition of other software components. A system/component is the result of the composition of many components at different levels.

Composition as a technique in object oriented design can be used to make and view a design at different levels and helps to design reusable components.

Levels of Reuse:

Design patterns enforce reuse at design level, not at code level. Software components insure reuse at code level. Also, frameworks insure code-level reuse and design patterns are implemented in the frameworks.

Current Static Design pattern Composition Approach:

Keller and Schauer (Keller and Schauer, 1997) proposed static composition for design patterns to create reusable components. It provides a way to create components if the user wants to follow some specific patterns. The downside is that it does not provide a way to predict the runtime behaviour of the generated components. More over, because of the lack of runtime behaviour analysis, some better alternatives are not considered in the design level.

Using proper design patterns is important for a good design. But according to some OO methodologies [Jacobson, 1992] a design mostly starts with use case analysis, not with the structural composition of design patterns as proposed by Keller and Schauer [1]. The use case approach is especially suitable to real-time software, where run-time behaviour and performance play an important role. For example, architecture design for network protocols is very complex, and many alternative designs are possible. Good architecture really depends on the application requirements and comes from an evolutionary process. The evolution of Conduit[2], Conduit+[3,4], and Java based framework[5,6] shows that a good design comes from continuous reuse and reassessment of available design patterns, and evolve the design with new design patterns which redistribute proper responsibilities to components in the architecture. The evolution goes hand in hand with careful use case

analysis, and is not based only on combining together available design patterns in a static manner.

A Possible Composition Approach with Use Case Design: A possible composition approach is to start with small use case and adopt a spiral approach assuming that some class library and/or 3rd-party components with some design patterns and/or framework are available.

A use case, or a scenario, or a use case map [Buhr and Casselman, 1996] is normally considered as a complete execution path through a software system. But doing use case analysis for component based design is difficult due to many layered encapsulations and the use of polymorphism. For performance analysis, bottom-up use case analysis is the easiest way to trace the performance at different levels.

The term *component* is used in COM/Activex, CORBA, and Javabeans as static reusable components. We distinguish between static components and operational (execution) components. Whereas static component is a piece of code (e.g., classes), an operational component represents the execution of an activity which provides a well defined functionality with a clear interface. **From our point of view, static software components can be built by grouping a number of operational components which correspond to different functionality performed by the same set of classes/objects.** This will be further described in Chapter 3.

2.2 Message Sequence Chart

Lots of work has been done in the literature for Message Sequence Chart leading to the Z.120 standard of the Telecommunication Standardization Sector of International Telecommunication Union (ITU-T, 1996). MSC was applied first to non-OO real-time system, then was imported into the OO methodologies, and is named now *sequence diagram* in UML[Fowler, et al., 1997].

According to Z.120, a Message Chart Sequence Chart (MSC) describes the sequences of messages interchanged between instances (an instance of an entity is an object which has the properties of this entity). Meta-language have been provided for textual grammar and graphical grammar. Message, gate, ordering, condition, timer, action, instance decomposition, loop, alternation are described in Z.120.

An MSC describes the communication between a number of system components, and between these components and the rest of world, called environment. For each system component covered by an MSC there is an instance axis. The communication between system components is performed by means of messages. The sending and consumption of messages are two asynchronous events. It is assumed that the environment of an MSC is capable of receiving and sending messages from and to the Message Sequence Chart; no ordering of message events within the environment is assumed. Although the behaviour of the environment is non-deterministic, it is assumed to obey the constraints given by the Message Sequence Chart.

There is no global time axis assumed for one Message Sequence Chart. Along each instance axis the time is running from top to bottom. but a proper time scale is not assumed. Total time ordering of events is assumed along each instance axis. Events of different instances are ordered via messages - a message must first be sent before it is consumed - or via the so called generalized ordering mechanism. A Message Sequence Chart imposes a partial ordering on the set of events being contained. A binary relation which is transitive, antisymmetric and reflexive is called partial order. This is illustrated in Figure 2.1.

Gates represents the interface between the MSC and its environment [ITU-T, 1996].

Gates may have explicit or implicit names.

Figure 2.1 Message Sequence Chart Ordering

The Message Sequence Chart defined in the ITU-T Z.120 standard is served for different purposes:

- (a) overview of a service as offered by several entities;
- (b) statement of requirements specification;
- (c) basis for elaboration of SDL specifications;
- (d) basis for system simulation and validation;
- (e) basis for selection and specification of test cases;
- (f) specification of communication;
- (g) interface specification;

(h) representation of use cases within object-oriented design and analysis.

We have to mention that some graphic syntax defined in UML is used in Figure 2.1. The Sequence Diagram defined in UML (pp. 103-112, Fowler and etc., 1997) is the standardization of the MSC for object oriented design. In a sequence diagram, the box at the top of a dashed vertical line describes an object which is similar to Z.120 definition except that a solid line is used in Z.120. The object's lifeline is the vertical line which represents the object's life during the interaction. This follows the "interaction diagram" used in OOSE (Jacobson, Christerson, Jonsson, and Overgaard, 1992). There are also definitions for Iteration (using * as syntax) and Condition (using [myCondition == "TRUE"]). But they are not as clear as the loop and condition defined in Z.120.

The Sequence Diagram defined in UML does not define an object to be called by nobody (Fowler and Scott, 1997). Traditionally object oriented system designers tend to use sequence diagram to describe a use case from end to end. The operations msg1, msg2, and msg4 in Figure 2.1 are called by other components which are unknown in the MSC. This feature is defined in Z120 possibly for the purpose of communication interface specification, but not defined in UML. We want to introduce this useful MSC feature to object oriented design because we need to create a generic recursive composition approach for both component oriented design and performance analysis. Modifications are required especially for outgoing messages (e.g., if :TypeA calls other object(s) outside of the msc exec_order after its msg1 is called). In Chapter 3 we will further describe the

component concept and how to conduct the use case composition for performance analysis.

Sequence diagram is the UML name for MSC. To avoid confusion, MSC is used in the thesis.

In MSC, a condition is described as a state. A condition may be defined for an object. A “*shared condition*” may be defined for a set of objects. A global condition referring to all objects in a MSC may be defined by means of the keyword *shared all*.

Weakness of MSC in Z.120:

- In MSC in Z120, all invocations are concrete. Polymorphism is one of the main key feature in object oriented design. It becomes extremely important when design patterns and framework are used in a design.
- There is no Message Sequence ID. We need to identify the message sequence in a MSC and an ID is straightforward to use and very useful in an analysis tool. (In UML, sequence number is used for Collaboration Diagrams only.)
- No performance information (execution time, probability of choices, etc.) is provided in a MSC.

From our point of view, a runtime component can be described by one or more Message Sequence Charts (MSCs). But the above limitation of the current MSC prevent us from efficiently using it for object oriented design (especially when design patterns and framework are used) and for performance analysis. Extension of the MSC specification is essential.

Use of HMSC/Activity Diagram

According to Fowler (1997) activity diagrams combine ideas from several techniques: the event diagrams of Jim Odell, SDL state modelling techniques, and Petri nets. It is particularly useful in connection with workflow and in describing behaviour that has a lot of parallel processing. The great disadvantage of activity diagram is that they do not make the links among actions and objects very clear. Some people feel that using activity diagrams is not object-oriented and thus not good.

However with the concept of operational component and abstract operational component in hand, we can consider each activity as an operational component with gates. All the operational components connected together with some special syntax's which are defined in UML, e.g., synchronization bars (fork and join), and decision activity can provide us with a complete high level use case specification.

Figure 2.2 Activity Diagram

From this point of view we can consider an activity diagram as an extension of the high-level MSC defined in Z.120. For example, synchronization bar is introduced for parallel processing, decision activity syntax is used instead of a condition syntax. Figure 2.2 shows a higher level of activity diagram abstracted from an example described by Fowler and Scott (1997).

Activity diagrams/HMSC can describe high-level use case and are very useful for representing multiple use cases at a higher level of abstraction, for understanding workflow across many use cases, and for dealing with concurrent processing. It was suggested (Fowler and Scott, 1997) not to use activity diagram to try to see how objects collaborate (an interaction diagram is recommended).

According to UML (Fowler and Scott, 1997), an activity is some task that needs to be done whether by a human or a computer; it is also a method on a class. In our view, an activity can be the result of multiple-object collaboration, especially when representing high-level abstractions. Informally, we can define an operational component as an activity with well defined interface. A more precise definition of operational component is given in section 3.1.2. As the focus of this research is for the performance data collection by use case composition for performance, we will concentrate on message sequence chart and operational components. The focus of the thesis is to aggregate performance measures (such as resource demands) at the level of each operational component. The concurrency and parallelism between operational components as shown in the activity diagram will be represented at the performance model level, and it is not described in the thesis.

2.3 Performance Analysis Algorithms

Smith summarized the Software Execution Models for procedure oriented design (Smith, Connie, 1990).

Execution graph notation and analysis algorithms are provided by Smith (1990). Sequential path reduction rule, repetition-path reduction rule, conditional-path reduction rule, copying rule, expanded node assignment rule, shortest and longest time computation rule, shortest path conditional reduction rule, longest-path conditional rule, and variance computation rules were proposed in her book (Smith, 1990).

2.3.1 Graph Analysis Algorithms

The basic structures of a control flow graph are sequence, loop, and case nodes. The algorithms apply iteratively reduction rules to translate a graph into a single node; then the average time, the maximum time, the minimum time, and variance for the original execution graph is the resulting time for the reduced graph.

2.3.1.1 Average time reduction

Sequential-Path Reduction Rule:

Figure 2.3 Sequential Path Reduction Rule

Repetition-Path Reduction Rule:

Figure 2.4 Repetition-Path Reduction Rule

Condition-Path Repetition:

Figure 2.5 Condition-Path Reduction Rule

2.3.1.2 Shortest and longest -time reduction

The only rule to change is the conditional-path rule. Others are similar to the average time reduction rule.

$$t_1^{i+1} = t_0^i + \min(t_j^i)$$

The shortest-path condition rule:

$$t_1^{i+1} = t_0^i + \max(t_j^i)$$

and the longest-path condition rule:

2.3.1.3 Variance reduction

Variance for Sequential-Path Reduction Rule:

$$V_1^{i+1} = \sum_{j=1}^n V_j^i$$

Variance for Repetition-Path Reduction Rule:

$$V_1^{i+1} = nV_1^i + t_1^2 n_v$$

where n is the loop repetition factor and n_v is its variance.

Variance for Conditional-Path Reduction Rule:

Define n as the number of conditional paths; I is an index set consisting of all combinations of the set for $\{1,2,3,...n\}$ taken at a time. There are $\binom{n}{2}$ such combinations.

$$\text{for } n=1, \quad V_1^{i+1} = V_0^i + V_1^i$$

$$\text{for } n \geq 2, \quad V_1^{i+1} = V_0^i + \sum_{j=1}^n p_j V_j^i + \sum_{j,k \in I} p_j p_k (t_j - t_k)^2$$

2.3.2 Issues about using Smith's work for performance analysis of object oriented designs:

- The graph notation is for procedure oriented design. Although it is possible to transfer an OO design to graph notation (Smith and Williams, 1997; Williams and

Smith, 1998), it is not natural and also not easy to do the translation from an object oriented design.

- There is no polymorphism in the analysis algorithms. Polymorphism is one of the main key feature in object oriented design. It becomes extremely important when design patterns and framework are used in a design. The execution behaviour of components in most object oriented systems is unknown until the complete scenario or use case is build. The current graph analysis algorithms are for concrete components (procedures). There is no concept of abstract components in the current algorithms.

In this thesis, MSC will be used directly instead of Smith's execution graph for analysis. Also, abstract runtime components and algorithms will be proposed to provide analysis algorithms for object oriented systems.

2.4 Network Protocol Design

Network protocols are complicated. A protocol stack is usually implemented with a layered software architecture as a result of the OSI standard architecture [Stallings, 1994]. For example, IP layer sits on top of Ethernet, or ATM AAL layer; TCP and UDP are on top of the IP layer. In the protocols, each layer is connected to the layer above and below it. Information (packets for low-level protocols, and message or other user information for high level protocols flows either from the user down to network or from the network up to the user. Layered architecture is not difficult to build, but reusability is not easy to

address without O-O design. The key problem is to factor out the common structure and behaviour from the protocol specific parts. The solution should be reusable software components (framework) which can be reused in different protocol implementations without modification of the framework.

Traditional procedural design has been successful for protocol implementation. It is easy to do performance analysis using the current graph analysis algorithms described by Smith (Smith, 1990). But reusability as the purpose of object oriented design and further component oriented design make the system so complicated that the performance analysis is very difficult.

2.4.1 Original Conduit Framework

The original “Conduit” framework was designed Zweig and Johnson (1990). That framework is a white-box style framework which provide the basic structure of conduits. In the Conduit framework, inheritance is heavily used for extension of a new protocol implementation. The whole design is not as reusable as the next versions, as explained in the following sections. These limitations results in more development and testing effort.

Figure 2.6 Class Diagram for Conduit (summarized from Zweig and Johnson, 1990)

The double dispatching pattern is used in the Conduit (Figure 2.7).

Figure 2.7 Double Dispatching Pattern in Conduit (summarized from Zweig, 1991)

The double-dispatching pattern ensures that each conduit has the opportunity to inspect the protocol-ID of the other to check that the requested connection is valid.

2.4.2 C++ Conduit+ Framework

Huni, Johnson, and Engel (1995) extended the white-box Conduit framework to build the black-box framework Conduits+. Conduit+ is a framework which employs a number of different design patterns. It separates Mux from real Conduit which makes the multiplex and demultiplex easier to implement; it adds ConduitFactory for dynamic allocation of new Conduits to a mux; and Visitor pattern is used to allow a Visitor to travel over a whole string of conduits before reaching its destination (conduit). Protocol implements a finite state machine.

The patterns are summarized in the followings:

- State Pattern: Protocols are implemented using the State pattern (pp.305, Gamma, Helm, Johnson, and Vlissides, 1995). Each state of the protocol is represented by a separate object. A Protocol delegates its behaviour to its state object, thus letting the protocol change its behaviour when its state changes. A protocol changes its state by replacing its old state-object with a new one which may be created by its old state (determined by the FSM).

- Singleton Pattern: Each protocol requires a new State class hierarchy with a new derived class for each state in the finite state machine. As there is at most one instance required for each State class, the State is designed using Singleton pattern (pp.127, Gamma, Helm, Johnson, and Vlissides, 1995).
- Command pattern: A Messenger contains the information chunk and has the apply() operation. It represents an event or an operation that is going to be invoked on the FSM of Protocols that it encounters as it passed through the graph of conduits. It is an example of Command pattern pp.233, Gamma, Helm, Johnson, and Vlissides, 1995)..

- Visitor

Pattern:

Figure 2.8 Visitor Pattern: MsgTransporter

A visitor arrives at a conduit when the `accept(:MsgTransporter)` is invoked by a previous conduit. The conduit then performs an operation `atProtocol(:Protocol,`

:State, :Conduit) on the visitor that indicates the class of the conduit. Thus, the visitor gets to know the conduit's subclass through the method that is invoked on it. It then apply its messenger on the protocol's current state object. A visitor can travel in either to network or from network directions.

- Strategy Pattern: An Accessor belongs to each mux. It computes the index of the sideB conduit to which the current InformationChunk should be dispatched; it also computes an index for a new sideB conduit. The mux handles an information chunk that arrives on sideA by using its Accessor to get an index, and then sending the information chunk out on the sideB conduit with that index.
- Prototype Pattern: A conduit factory must be parametrized by the kind of conduits that it creates. Proposed solution is for each conduit factory to have a prototypical conduit that it copies when it needs to create a new one. Using the Prototype pattern requires that conduits be able to copy themselves.
- Adaptor Pattern: The adaptor conduit is used to interface the framework to some other software or hardware.

The Conduit+ was implemented in C++ and was used in ATM layer 3 protocol by ASCOM TECH (Engel, Bieri, and Keller, 1995). The C++ Conduit+ framework as an industry product which is developed in ASCOM TECH has reached a stable and application-independent state(Engel, Bieri, and Keller, 1995).

2.4.3 Java Conduit Framework

Nikander, Parssinen, Sahlin, and Hoglund (1997) and Parssinen (1997) implemented the Conduit + in Java with some minor modifications:

- Session pattern (Beck and Johnson, 1994) is used. Sessions are the basic functional units in the framework instead of using the Protocol used in C++ Conduit+ framework. A session implements the finite state machine of a protocol or part of it (in this case, a larger protocol FSM is decomposed into some smaller FSMs). The use case for a Session is described in Figure 4.11.

Protocol Pattern is introduced as the composition of other conduits (Sessions, Protocols, Adaptors, ...). It is used purely for the easiness of different level component design and reuse. The Java Conduit Framework was developed in Helsinki University of Technology.

The stable version we got is Jacob2.

Figure 2.9 Class Hierarchy and Relationship of Jacob2 Framework

Figure 2.9 shows the class hierarchy and relationship. The basics of the classes and interfaces are described in the followings:

- *Conduit* is an interface which is purely abstract. *BaseConduit* is the base class for all the different protocols. *BaseConduit* is abstract, but its operation `accept(:Visitor)` is final. This operation is the only runtime operation for any *Conduit* to accept a *Visitor*. The subclasses are:

Session is the basic functional units of the framework. A session implements the finite state machine of a protocol, or some aspects of it. The session remembers the state of the communication, and delegates the implementation of the protocol to a number of *State* objects, using the *State* pattern. It is an abstract class.

Protocol is a kind of meta conduit that encapsulates several other conduits. A *Protocol* has side A and side B. *Protocol* can be used as a component to build more complex *Protocols*. It is an abstract class.

Adaptors are used to connect the conduit graph to the outside world. It has only side A. It is an abstract class.

A *Mux* is a conduit that connects one side A conduit to any number of side B conduits. It is used to multiplex and demultiplex protocol messages.

ConduitFactory is typically connected to side B of a *Mux*. A *ConduitFactory* can create a new *Session/Protocol* and installs it to the side B of *Mux*. Both of them are final classes. As *Mux* is designed as a blackbox, whitebox-style *Accessor* interface is used with *Mux* which follows the strategy pattern.

- *Visitor* is an interface which is purely abstract. All *Visitors* traverse the conduit graph carrying something. A *Transporter* is an abstract class who implements the

Visitor interface. Every time a Transporter arrives to a conduit, it continues to the other side of the conduit. The direction may be redirected by Conduits; the Transporter just walks blindly. *MessageTransporter* is a subclass of Transporter that carries Messages.

- *Messenger* is a Java interface that acts as a place-holder, separator, between conduit graph and actual protocol functionality that is implemented as States, Sessions, and Accessors(if it applies).
- *Message* contains MessageBuffer, MessageTransporter, and Messenger.
- *State* is an interface. Each protocol implementation must provides its own set of State classes. Typically all State classes use the Singleton pattern.

To instantiate the Jacob framework to build a new application, all the interfaces and abstract classes, and abstract operations in the framework must be implemented in new concrete subclasses. For example, new concrete subclasses/implementation classes for Session, Protocol, Accessor, Messenger, and State have to be implemented for the application. And it is quite possible the user may want to define a new Message class or a new Transporter as subclasses due to the requirement of the application.

The Java Conduit framework will be used for the development of the case study and performance measurement in the thesis.

Our purpose is to use our methodology to integrate design and performance analysis at the early stage of a project by doing a detailed application design as case study. We will complete the coding for the case study, and measure its performance to collect

performance information. There is no intention to judge the performance of the framework itself.

2.5 Summary

Difficulties to combining O-O use-case design with performance analysis:

- The current composition approach is by structure, but for performance analysis the composition by behaviour is essential.
- The Message Sequence Chart may be used for behaviour composition, but performance information is required in a MSC.
- The performance analysis algorithms are available as graph theory and execution model for procedural design, but they are weak for the design and performance analysis of object oriented systems.

How to adopt use case composition and combine performance analysis with object-oriented design in an early design stage? This is what the thesis will try to achieve.

Use case composition at runtime components level, extension of the current Message Sequence Chart, and proper performance analysis rules for object oriented systems are the main target of this research. A detailed design which combine the use case composition and performance analysis at the design stage will be conducted using the Jacob Conduit Framework as the case study. Coding and Performance measurement will be conducted for the case study.

NOTE TO USERS

Page(s) missing in number only; text follows. Page(s) were microfilmed as received.

UMI

Chapter 3 Use Case Composition Rule for O-O Design

Use cases elicit requirements from the users in meaningful chunks. Interaction diagrams show how several objects collaborate in use cases. There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. The sequence diagrams emphasize the execution sequence and make it easy to see the order in which actions occur. Collaboration diagram indicates how objects are connected and their exchange of messages. A sequence diagram is a Message Sequence Chart (MSC) for object oriented design. However, current sequence diagrams as defined in UML do not have some important functionalities of MSC (e.g., the gate concept which can be borrowed for component oriented design.) We will describe it in the chapter. To avoid confusion, in this thesis, our variety of message sequence chart is used to describe the runtime behaviour of an object oriented design for performance analysis.

3.1 Concepts

3.1.1 Extension to Component and Interface Concepts

Traditionally software components have been available in the form of procedures and functions for numerical and statistical applications (especially in FORTRAN).

The definition of 'component' was formulated at the ECOOP in 1996 (Szyperski and Pfister, 1997) as:

'A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.'

COM/Activex[Platt, 1997], Corba[Eichner et al., 1997; Orfali, et al., 1998], and Javabeans[Calder, et al., 1998] have been recognized as commercial component technology. In this research, there is no intention to focus on any specific technology. The focus is on runtime behaviour for object-oriented design and further component-oriented design in a general sense.

Interfaces are defined in COM/Activex, Corba, and Javabeans. A component contains one or more interfaces. Each interface contains one to many operations and may contain some properties. CORBA follows a traditional object model. An object has a single interface, although this interface may be composed of other interfaces using multiple interface inheritance. The actual interface provided may be a subtype of the interface expected. The additional capabilities can be explored dynamically. COM has immutable interfaces that, once published, cannot be extended or modified. But a COM object can have multiple interfaces. The set of provided interfaces can vary and can be explored dynamically. A

Java object can also implement multiple interfaces, although this is closer to multiple interface inheritance than to COM's totally separate interfaces.

The interface concept is weak in C++, Smalltalk, and most of other languages. To make it generic, we imagine that there is one interface "public" for C++, and Smalltalk objects which contains all the public methods and attributes. This extension is helpful to produce generic methodology for use case composition. As we will see that our methodology applies to both the industry recognized components and the objects in object-oriented design.

3.1.2 Proposed Basic Structures

In this research, we propose the concept of operational component which uses previously introduced concepts of gate and message.

An *operational component* is a reusable entity that interacts with its surroundings through one or more gates at runtime. From the code point of view, an operational component involves one or more basic objects (C++, Java, etc.) and maybe other operational components. The runtime behaviour of a system can be described by the interactions between its operational components and those between its components and the surroundings of the system.

Details are described latter in this section on the type of operational components. In the next Chapter, we will show that it is possible to group several related operational

components into one static software component (which is actually the current “component” concept defined in industry).

A *gate* is a part of an operational component that describes the interaction of the operational component with the outside world, and implies a well defined operation and specific interface (for a class, COM/Activex, Corba, or Java). Each gate of an operational component is associated with a specific message and plays a particular role in a collaboration that the operational component has with other objects/operational components. A gate is a contract between operational components (and static software component too, as an operational component describes the behaviour of a static software component).

A *message* is a physical operation of an interface which is tied with a gate of the operational component, and a gate of another operational component. Throughout this thesis, we consider that messages are implemented as operation calls, which means that the operational components are executed sequentially.

The name of a gate is given implicitly by the direction of the message through the gate and the message name. In a sequence diagram, the name of a gate is usually not specified. But it is important to keep this concept in mind when doing a design. The outgoing message has not been used in traditional sequence diagram in object oriented design (this is because the lack of gate concept in O-O design and the fact that designers tend to provide a complete use case when using sequence diagrams).

Figure 3.1 Structure Definition of Basic Concepts

In this research we define two types of gates which are important for composition. One is the in-gate in an operational component, which is used for connection by other operational component. The other is the out-gate in an operational component which is used for the operational component itself to connect to (i.e., call) other operational components.

The outgoingMessage,<OpComponentC> in Figure 3.1 shows the contract between the operational component OpComponentA with the operational component OpComponentC. This means that its out-gate has to be plugged into the out-gate of the operational component OpComponentC. We propose this concept for the composition of operational components. More clear identification of gate is described in Figure 3.6 (section 3.2.1.1).

Figure 3.2 Collaboration Diagrams for Gate Connection

Figure 3.2 shows the synchronous operation call. When two operational components are connected together (Figure 3.2), it is similar to a hardware connection. Everything inside OpCompA is a black box from the point of view of box OpCompB. What the box OpcompB is interested in is the out-port in box OpCompA only. To connect to box OpCompB, the only interface box OpCompA needs to know is that box OpcompB has a compatible in-port. The message myMsg is a physical operation of OpCompB. It is interesting to see that the out-gate of OpCompA and the in-gate of OpCompB are associated with the message myMsg, and it is the message who builds up the connection between the two gates of two separate operational components.

To be consistent with the notion of interface in Component Software, we will consider that an interface consists of one or more gates. Each gate is associated with a connection (an operation) between objects. An object may have one or more interfaces. The operations are the public methods in C++ and Java, and interface methods in COM/Activex, Corba, and Java which can be accessed by other components. In O-O

design or component oriented design, the interfaces are among the most important things that an architecture designer has to consider.

Types of Operational component

An operational component may contain one or more operational components. The “contain” relationship of operational components can be recursive.

Regarding the level of complexity, there are two kinds of operational components. One is the “basic operational component”, which contains *basic objects* (i.e., C++ objects, Java objects, 3rd-party COM/Activex or Javabeans or Corba components) which is the lowest level operational component the designer is interested in. (The 3rd party components or other base objects are considered as base component. The design is not going to refine or develop them any further). The other one is the “composite operational component” which may be composed of *basic objects*, basic operational components and/or composite operational components. In this research, the “operational component” normally refer to composite operational component, but sometimes to basic operational component as well. The terms “basic operational component” and “composite operational component” are used in places where the knowledge of differentiation is required.

For object oriented design, or further component oriented design, an operational component can be either concrete or abstract. A concrete operational component is an operational component in which no dynamic binding exists at the current view of the design level. An abstract operational component is a kind of operational component whose runtime behaviour depends on dynamic binding at the runtime (which depends on

the input to the operational component). This is the very basic behaviour of object oriented design and further component oriented design, especially when design patterns and framework are used. In performance analysis, it is important to know if an operational component is concrete or abstract. Abstract operational components have dynamic performance behaviour which depends on runtime late binding. Concrete operational components do not have late binding as known at the analysis level. Although the performance of concrete operational component may depends on its arguments passed over at the runtime, no late binding is involved.

3.1.3 Extension Required for MSC and Performance Analysis Algorithms

According to Z.120 defined by ITU-T, a Message Sequence Chart (MSC) shows the sequences of messages interchanged between system components and their environment we will use. MSCs can be used to describe an operational component. In the case of real-time systems, MSCs are very powerful for design and for the prediction of system performance. For performance analysis, we propose to add to MSC message sequence ID, and some performance information (such as CPU demands and choice probability for alternations). Details are described in section 3.2 of this chapter.

In MSCs and AMSCs (Abstract MSC, which will be described in next section), the gate notation is ignored for simplicity. Only the messages going in and out of the gates are

displayed. When there is a message going out of an operational component, we assume that there is a gate to which the message is directed.

An abstract operational component may be described by an Abstract Message Sequence Chart (AMSC). The description of abstract operational component and AMSC is in the section 3.2.4.

3.2 Extension of Message Sequence Chart

Message sequence charts (MSC) are standardized by International Telecommunication Union (ITU_T) in Z.120. For performance analysis, extensions to MSC (such as the use of message sequence ID, the Abstract Message Sequence Chart, etc.) are proposed in this section. The analysis algorithms will be described in the next section.

3.2.1 Message Sequence ID

MSC in Z120 does not provide the exact order of messages. To address the messaging sequence in a MSC, we propose a sequence ID. The following requirements have been considered to adopt a numbering scheme:

- to be able to identify the sequence in time domain
- to be able to identify the interface method name
- to be able to identify all the gates in all the operational components.

The sequence ID is defined as

“sequenceNumber,operation[,numberOfAccess][,<nextObject[.interface]>]”.

The *sequenceNumber* can be numbered using either a simple numbering scheme or a hierarchy scheme. The simple numbering scheme is widely used as it is more clear in what regards the overall sequence; the hierarchy scheme indicates more clearly which operation is calling which other operation.

An operation is represented by an *operation name*. It is the public method or interface method of a class of C++ or Java or an interface method of Activex/COM or Java.

The *numberOfAccess* is the sequence number of a call to an operational component from another operational component. This entry is optional. It is defined to make it easy to understand the behavior of interaction between different operational components.

The *nextObject* is the object to which the message is sent to (can be a basic object or an operational component).

The *“interface”* is the interface of *nextObject* which the outgoing operation belongs to. It is optional (for C++, only public can be used). *<nextObject[.interface]>* shows the exact interface the operation is contracted between the current operational component and the next operational component which the outgoing operation belongs to.

The complete information provided by *“outGoingObject[.interface].operation”* indicates the exact out-gate from the operational component. To plug this operational component to another operational component at the out-gate, it is necessary that the other operational component provide the same type of in-gate.

3.2.1.1 Message Sequence ID: Hierarchy Structure

Figure 3.3 Message Sequence Chart Using Hierarchy Numbering Structure

In this chart, the operational component consists of basic operational components (the lowest level of operational components) BasicSub1, BasicSub2, BasicSub3, and another operational component Subsystem1. The composed operational component (MySubsystem) is described in Figure 3.4. The in-gate is x,name, while the out-gate is x.1,name7,<exeComponent.interfaceNm>.

Figure 3.4 Composition of MySubsystem with Hierarchy Numbering Scheme

Figure 3.5 Message Sequence Chart Using Simple Numbering

3.2.1.2 Message Sequence ID: Simple Numbering

This is the same operational component as that described in Figure 3.3. The composed operational component MySubsystem is described in Figure 3.6. The in-gate message is 1,name, while the out-gate message is 2,name7,<exeComponent.interfaceNm>.

Figure 3.6 Composition of MySubsystem with Simple Numbering Scheme

The difference between the hierarchy-structure numbering and simple numbering is that: although the later is simpler but the former indicates how many objects/operational components have been involved in an operation. The hierarchy structure numbering is preferable when performance analysis is conducted, and is chosen for design.

3.2.2 Add Performance Information for Alternation and Loop in MSC

3.2.2.1 Alternation

The probability of choosing each case in an alternation is required for performance analysis. This information is missing in the Z.120. It is proposed to add this information to the MSC were suitable.

Figure 3.7 Performance Information in Alternation

3.2.2.2 Loop

Figure 3.8 Loop with n Repetitions

In Figure 3.8, the number of repetition is deterministic. In reality, loop<0,n> and loop<infinite> could be used in a design especially for real-time systems. A discussion can be found in section 3.3.2.

3.2.3 Composition of Message Sequence Chart in a Design

A good architecture design relies on proper operational component design. Design patterns and framework can be used to design pluggable operational components.

Composition and decomposition are useful to make the design understandable at different levels of abstraction.

(1) Composition

Composition is widely used in use case design to hide detailed knowledge at a higher level of abstraction and build operational components at different granularity levels. It is a powerful approach to compose operational components recursively. A lot of knowledge about design patterns and framework are required in order to use composition for building highly reusable operational components. After composition, the only thing the outside users have to know about the composed operational component is its gate information. From this point of view, it becomes a black box to other components.

(2) Decomposition

Decomposition allows us to see a more detailed picture of a system (component). As decomposition can be used at very different levels and on different operational components, it is very useful to the designers especially when applying a top-down design approach.

3.2.4 Abstract MSC and Abstract Operational Component

A lot of software systems using object oriented design are hard to analyze, especially for their performance characteristics due to the use of dynamic binding (or late binding). Most of these systems contains some software patterns/frameworks which are almost a mystery until a designer walks through detailed runtime use cases. “Abstract Operational component” is defined as a kind of operational component whose behaviour depends on dynamic/late binding of the objects/operational components it contains at the runtime. In other words, AMSCs are used to describe an abstract operational component.

For MSC we propose a new concept for object-oriented design, that is, Abstract MSC (AMSC). Figure 3.9 shows a AMSC.

Figure 3.9 A simple Abstract MSC

In Figure 3.9, the operation OP1 is an abstract function. Its actual behaviour is unknown until the type of BaseObjectType is finalized to any of the concrete types at runtime. For example, there are several concrete classes (ConcreteObj1, ConcreteObj2, ConcreteObj3, etc.) who are subclass of BaseObjectType. All of the concrete classes have their own implementation of the abstract function OP1. In C++ implementation, the abstractOP1 is a virtual function, and the real runtime behaviour of the system depends on what the pointer of type BaseObjectType points to.

Figure 3.10 A simple Concrete MSC

Figure 3.10 shows one of the concrete MSC for the AMSC in Figure 3.10. It is interesting to know that, for object oriented design the real behaviour of an abstract operational component which is represented by an AMSC is a function of the way in which the abstraction is instantiated. (Of course, apart from that, like procedure oriented design, it could be a function of the arguments in OP1). The possible values to which the abstraction can be instantiated may be determined at the design time when all the

operational components are connected together to build an end-to-end complete use case or may be determined from the user input.

The abstract operational component and AMSC concepts are important in object oriented design and for its performance analysis. Without the differentiation of AMSC from concrete MSC (CMSC), and that of abstract operational component from concrete operational component, it is hard to tell what is the real-time behaviour of the system at the runtime. Later in Chapter 4, we will see that this is usually helpful for an OO implementation network protocols based on a given framework.

In the following context, AMSC represents abstract message sequence chart, whereas the MSC/CMSC represents concrete message sequence chart according to the traditional MSC concept.

The performance analysis algorithms for both concrete and abstract MSCs are provided in the next section.

3.3 Performance Analysis Algorithms for Use Case Composition

The graph analysis algorithm has been summarized by Cornnie U. Smith for procedural oriented design ([21], Smith, 1990). In this research, we will extend it to object oriented design.

During the research, we found a paper of Smith and Williams (Smith and Williams, 1997) about applying the graph analysis algorithms to object oriented design. In the paper, the object oriented design was converted to the traditional Execution Graph for performance analysis which is a different way from our current direct analysis approach with MSC. With Execution Graph, the paper did not provide a methodology to consider the dynamic/late binding behaviour of object oriented design.

In procedural oriented design, procedures/functions are the basic units to consider; while in object oriented design, objects and/or operational components used in the thesis (both simple and composite operational components) are the basic units of the analysis. In performance analysis, average/shortest/longest access time and its variance to an operational component is used.

Similar to Smith's consideration (pp. 175, Smith, 1990), the term "time" is used in a generic sense in the following sections.

3.3.1 Rules for Sequential MSC

3.3.1.1 Basic operational component

A basic operational component is the lowest level of operational component. Figure 3.11 shows the detailed view and composition view of a basic operational component SimpleSubA. *:c*, *:d*, and *:e* are instances of basic type *c*, *d*, and *e* respectively. *c*, *d*, and *e* are C++ or Java classes, 3rd-party COM/Activex, Javabeans, or Corba components.

For object oriented design (including component oriented design), we define

- $t_{ave, c.op}$ as the average time for the operation op of an operational component/object :c;
- $t_{min, c.op}$ as the minimum time for the operation op of an operational component/object :c;
- $t_{max, c.op}$ as the maximum for the operation op of an operational component/object :c;
- $V_{c.op}$ as the variance for the operation op of an operational component/object :c.

It is assumed that the t variables are stochastic independent.

The lowest level performance information (average time, variance, and etc.) is defined at the lowest granularity level at which the performance analysis will be conducted.

Figure 3.11. Basic Operational Component Composition Rule

The sequence ID defined in section 3.2.1 is:

“sequenceNumber,operation[,numberOfAccess][<nextObject[.interface]>]”.

For Figure 3.11, we use one variance of the above definition as:

“sequenceNumber,operation[,nextObject]”.

Assuming that the sequence ID for an operation op1 on object :Obj is y.z,op (here y and z are integers, e.g., 1, 2, ...) , then the average time for that operation is the aggregation described in the following:

$$t_{ave,Obj.op1} = \sum_{i \in S} \sum_{op \in M_i} t_{ave,i.op} \Big|_{seq \text{ start with } y.z \text{ and with only one more digit}} + t_{ave,Obj.op1.INTERNAL} \quad (3-1)$$

where S is the set which contains all the objects in the operational component except the :Obj; M_i is the set which contains all the operations for object i ; the seq is in hierarchy scheme (e.g., y.z.1). The condition for seq in Eq. 3-1 suggests that only the object.operations which are called by object :Obj as the result of Obj.op1 operation are chosen. The $t_{ave,Obj.op1.INTERNAL}$ is the average time :Obj spends for the call Obj.op1 internally.

The maximum and minimum times are aggregated in the same way as that in Eq. 3-1.

Similarly the variance for the operation op1 on object :Obj is the aggregation described in the following:

$$V_{Obj.op1} = \sum_{i \in S} \sum_{op \in M_i} V_{i.op} \Big|_{seq \text{ start with } y.z \text{ and with only one more digit}} + V_{Obj.op1.INTERNAL} \quad (3-2)$$

For example, the times for Fig. 3.11 are:

- Average time for e.nm3():

$$t_{ave,e.nm3} = t_{ave,exCompA.nm4} + t_{ave,e.nm3.INTERNAL} \quad (3-3)$$

- Average time for c.nm():

$$t_{ave,c.nm} = t_{ave,d.nm1} + t_{ave,d.nm2} + t_{ave,e.nm3} + t_{ave,c.nm.INTERNAL} \quad (3-4)$$

- Variance for e.nm3():

$$V_{e.nm3} = V_{exCompA.nm4} + V_{e.nm3.INTERNAL} \quad (3-5)$$

- Variance for c.nm():

$$V_{c.nm} = V_{d.nm1} + V_{d.nm2} + V_{e.nm3} + V_{c.nm.INTERNAL} \quad (3-6)$$

3.3.1.2 Operational components within other operational component

An operational component may contain one or more operational components and objects of basic objects. Figure 3.12 shows the detailed view and composition view of an operational component CompoSubA. “:f”, “:SimpleSubA”, and “:g” are instances of basic type f, operational component SimpleSubA, and basic type g respectively.

Figure 3.12. Composition Rule: Operational Component within Operational Component

The rule derived in section 3.3.1.1 applies to CompoSubA in Figure 3.12. Here we assume that the SimpleSubA is the SimpleSubA described in Figure 3.11. Then the average time for CompoSubA is calculated as:

$$t_{ave, CompoSubA.nm0} = t_{ave, f.nm0} = t_{ave, SimpleSubA.nm} + t_{ave, f.nm0.INTERNAL} \quad (3-7)$$

If the designer wants to see a relationship in which the SimpleSubA is viewed as a whitebox, Eq. (3-3) can be easily substituted in the above relationship to obtain more detailed information. The result is not shown here.

The variance for CompoSubA is calculated as:

$$V_{CompoSubA.nm0} = V_{f.nm0} = V_{SimpleSubA.nm} + V_{f.nm0.INTERNAL} \quad (3-8)$$

3.3.2 Rules for Loop in MSC

:c, :d, :e are operational components and/or basic objects. The following rules apply recursively.

Figure 3.13. Loop Operational Component Composition Rule

Syntax $\langle m, n \rangle$ is defined here to describe the fact of looping with iteration number between m and n ($m \leq n$). The number of repetition is k whose value is between m and n and depends on probability distribution of the number of repetition for an application. Here k is considered as known, and its variance is also known as V_k .

The average time for LoopSubA.nm is calculated as:

$$t_{ave, LoopSubA.loopop1} = k(t_{ave, c.nm} + t_{ave, d.nm1} + t_{ave, e.nm2}) \quad (3-9)$$

The variance for LoopSubA is calculated as:

$$V_{\text{LoopSubA.loopop1}} = k(V_{c.nm} + V_{d.nm1} + V_{e.nm2}) + V_k(t_{\text{ave},c.nm}^2 + t_{\text{ave},d.nm1}^2 + t_{\text{ave},e.nm2}^2) \quad (3-10)$$

If k is deterministic, then the Eq. (3-10) is identical to the sequential composition rule.

3.3.3 Rules for Alternation in MSC

Let c , d , e be operational components and/or basic objects. The following rules should apply recursively.

The syntax of alternation in MSC is shown in Fig. 3.14, where p_x is the probability of message $x, nm1$ and p_y is the probability of the alternate message $y, nm2$.

Figure 3.14. Alternation Operational Component Composition Rule

The average time for AlterSubA.altop1 is calculated as:

$$t_{ave,AlterSubA.altop1} = p_x t_{ave,c.nm1} + p_y t_{ave,d.nm2} \quad (3-11)$$

The variance for AlterSubA.alt is calculated as:

$$V_{AlterSubA.altop1} = p_x V_{c.nm1} + p_y V_{d.nm2} + p_x p_y (t_{ave,c.nm1} + t_{ave,d.nm2})^2 \quad (3-12)$$

The composed operational component AlterSubA has only one operation alt.

Note: For performance analysis, it is quite possible that the most frequent case is selected.

Then the probability for the selected one is 1, while that for other alternations is 0.

3.3.4 Rules for AMSC

AMSC is a special MSC in which the accurate behaviour of objects/operational components inside the AMSC is dynamic, and is determined by late binding. “Good” OO design promotes the use of polymorphism which leads to the use of AMSCs. As we described earlier, concrete operational components were illustrated by concrete MSCs. In this section, an abstract operational component is illustrated by AMSC.

In the following, a simple AMSC corresponding to a simple design for user interface design is described. Figure 3.15 shows the class hierarchy of operational component AMSCSubA.

Figure 3.15 Class Hierarchy of An AMSC Example

Figure 3.16 shows the Abstract message sequence chart (AMSC) of the abstract operational component AMSCSubA.

Figure 3.16 Abstract Operational Component AMSCSubA

Then the average time for AMSCSubA is calculated as:

$$t_{ave,AMSCSubA.Do} = t_{ave,CAgent.Do} = t_{ave, DoBase.Do} + t_{ave,CAgent.Do.INTERNAL} \quad (3-13)$$

where the quantities on the right hand side are unknown until the abstract classes are instantiated to concrete objects. This is a generic formula which will be applied to all the concrete MSCs instantiated from this AMSC by substituting each term at the right side with the concrete values of the concrete MSCs.

The variance for AMSCSubA is calculated as:

$$V_{AMSCSubA.Do} = V_{CAgent.Do} = V_{CDoBase.Do} + V_{CAgent.Do.INTERNAL} \quad (3-14)$$

It is noticed that the values of time and variance for CDoBase.Do are unknown in an AMSC until the exact type of the instantiated objects involved in the component are known; usually, these types depends on the input to CAgent. Therefore, the parameters including the average time, maximum time, minimum time, and variance of AMSCSubA are functions of the input to the AMSC. However once the input is determined, the abstract MSC becomes concrete, and its exact behaviour is determined. In this example as a part inside a scenario, an abstract operational component is a component whose behaviour depends on the input to itself. A scenario consists of operational components who are connected together to form a complete end-to-end use case. For a complete scenario (use case), the inputs to each operational component are known, thus the Abstract operational components become concrete operational components.

From an implementation point of view, the AMSCSubA may be implemented as a software component or a wrapper class with operation Do(); it can also be totally abstract in which case, the Do operation for AMSCSubA corresponds to the physical implementation of CAgent.Do().

An abstract operational component (e.g., AMSCSubA) is the abstraction of a set of concrete operational components to which it can be instantiated. All the concrete operational components have similar behaviour, the AMSC “generic” formulas for execution times (such as Eqs. 3-13 and 3-14) can be applied.

In a framework-based design, abstract operational components may come with the framework, and the designers using the framework has to implement concrete classes which define all the abstract operations. In this way, an abstract operational component will generate a set of concrete components.

The concepts of abstract operational component and AMSC help performance predictions, as well. Once the concrete implementations of the polymorphic functions have been decided and their execution times estimated, the “generic” formulas for AMSC can be applied.

3.4 Summary

In this Chapter, we have proposed two new concepts of *Operational Component*, and *abstract MSC* (AMSC), which are useful both for design and for performance analysis.’

Message sequence chart have been extended to include performance information.

Composition rules for sequential composition, loop composition, and alternation composition have been adapted for OO operational components from similar work for procedure-oriented design (Smith, 1990).

A detailed case study will be conducted in Chapter 4 by using a known framework for building network protocols.

NOTE TO USERS

Page(s) missing in number only; text follows. Page(s) were microfilmed as received.

UMI

Chapter 4 Case Study: Implementation of An Application Protocol Using the Java Conduit Framework

The case study presented in this chapter implemented an application protocol by using Java Conduit Framework Jacob2 created by Nikander and Parssinen (1997). Coding and debugging were done using VisualAge Java on Windows NT.

4.1 System and Architecture Design

4.1.1 Use Cases

The highest level use case for an application protocol is plotted in Figure 4.1.

(a) Send a message

(b) Receive a message

Figure 4.1 High level use case for an application

In the use case “Send Message”, the application data structure presented in an application format (e.g., an instance of a class) is converted first to a string format; then the content of the message is encoded with a specific protocol (e.g. E164) and is sent to the lower protocol layer (e.g., TCP, UDP). Hence the “Send Message” use case can be decomposed into two use cases as described in Figure 4.2.

4.2 Decomposed "Send Message" use case

In the use case "Receive Message", the encoded message received from the lower layer (e.g. TCP) is decoded; then it is converted to the application format.

4.3 Decomposed "Receive Message" use case

4.1.2 Analysis and Design

From the above use cases, we define AppMessage as an information object type in the system. Other objects will be described later.

4.1.2.1 Mapping from Use Case to Operational Components

It is noticed from Figures 4.2 and 4.3 that:

- (1) the use cases “Encode message” and “Decode message” handle messages using the same coding protocol. This observation suggests that although the two use cases belong to different higher level use cases, they may be merged in the design, as they use the same related data (message) and the same rules for processing data.
- use cases “Convert from string to app format” and “Convert from app format to string format” have the same application format and string format. Thus they may be merged in the design, as they use the same data (message) and the same rules for processing data.

The class design will be described in section 4.1.2.2. Here we will discuss how to design operational components from the use case analysis.

The use cases are plotted together in Figure 4.4.

4.4 Combined use cases

We will map the use cases into operational components as follows:

- The use case “Convert app format to string format” is mapped to the operational component `CompositeStructSessionToNet`.
- The use case “Encode message” is mapped to the operational component `CompositeCodeSessionToNet`.
- The use case “Convert string format to app format” is mapped to the operational component `CompositeStructSessionFromNet`.
- The use case “Decode message” is mapped to the operational component `CompositeCodeSessionFromNet`.
- The high level use case in Figure 4.2 (which is the left part of Figure 4.4) is mapped to the operational component `CompositeAppProtocolToNet`. It is a composite of the operational components `CompositeStructSessionToNet` and `CompositeCodeSessionToNet`.

- The high level use case in Figure 4.3 (which is the right part of Figure 4.4) is mapped to the operational component `CompositeAppProtocolFromNet`. It is a composite of the operational components `CompositeStructSessionFromNet` and `CompositeCodeSessionFromNet`.

We may draw the mapped operational components in Figure 4.5. The `TopAdapter` and `BottomAdapter` are plotted to describe connections to the upper and lower layers which are not our focus in this example.

Figure 4.5 Complete Picture of Message Flowing in Two Directions

As we discussed earlier in this section, from object orientation point of view, CompositeStructSessionToNet and CompositeStructSessionFromNet may be abstracted to an abstract operational component CompositeStructSession. CompositeCodeSessionToNet and CompositeCodeSessionFromNet may be abstracted to an abstract operational component CompositeCodeSession. Also CompositeAppProtocolToNet and CompositeAppProtocolFromNet may be grouped into an abstract operational component CompositeAppProtocol. The abstract operational components can be used at higher level use case design, and can be useful for the logical structure design of an object oriented system.

4.1.2.2 Mapping from Operational Components to Classes

By looking at Figure 4.5, and by knowing the Conduit Framework Jacob2 framework, it is easy to realize that the concrete operational components CompositeCodeSessionToNet, CompositeCodeSessionToNet, CompositeCodeSessionFromNet, and CompositeCodeSessionFromNet can be implemented directly using the Session pattern. Also the high level concrete operational components CompositeAppProtocolFromNet and CompositeAppProtocolToNet directly using the Protocol pattern used in Jacob2. By using the Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1994; Nikander, Jarssinen, Sahlim, and Hoglund, 1997), the only operation is accept(:Visitor) where the Visitor is an interface defined in Jacob2.

The mapping from operations operational components to classes are discussed in the following:

- Both CompositeStructSessionFromNet and CompositeStructSessionToNet

Figure 4.6 Operational Components related to StructSession

(concrete operational components) are not mapped to individual classes directly. Instead, the abstract operational component `CompositeStructSession` which is an abstraction of the two concrete operational components is mapped to class `StructSession` in coding. This is because that they share the same object types (`AppMessage`, `StructState`, and `MessageTransporter`) and similar rules, and we want to minimize the code. For `StructSession`, the consideration is similar (we will not repeat the similar word). One abstract operational component maps into exact one class in this case study. Mapping from abstract operational component instead of concrete operational components helps to optimize class structures, and

further helps to feed the gap between use case design and object orientation. Mapping from concrete operational components to classes may results in more classes in a design which is not good from object orientation point of view because proper information sharing and some basic internal operations' sharing may not happen.

- The abstract operational component of these two concrete operational components is mapped into class CodeSession. Similar to StructSession, both CompositeCodeSessionFromNet and CompositeCodeSessionToNet are not mapped to a single class implementation each.

Figure 4.7 Operational Components related to CodeSession

- Class AppProtocol is the implementation of the abstract operational component CompositeAppProtocol. Both ComposedAppProtocolToNet and ComposedAppProtocolFromNet are not directly implemented to individual

(a) Concrete Operational Components

(b) Abstract Operational Component

Figure 4.8 Operational Components related to AppProtocol

classes each.

The above implementation classes are displayed in Figure 4.9.

Figure 4.9 AppProtocol Conduit

A and *B* are of type BaseConduit which is the abstract class who implements Conduit interface. The double rectangle in Figure 4.9 includes both StructSession and CodeSession as one unit which is AppProtocol. This unit corresponds to the Protocol pattern in Jacob2. It encapsulates the application protocol related conduits.

The class hierarchy for our implementation is in Figure 4.10. We have described the classes which are important in the high level design. Other classes are the results of the implementation following the Jacob2 framework. All of them will appear in the individual

detailed design in the following sections. All the Jacob2 classes and interfaces are packaged as conduits package while all of our implementation classes are packaged as AppConduit package.

4.2 Design of Operational Components: Sessions

First of all, the Session pattern in Jacob2 framework is described here before we use it for our detailed design.

Figure 4.11 The highest level of AMSC for Session based on Jacob2

Figure 4.11 shows a high level Abstract Message Sequence Chart of Java Conduit when the Visitor arrives at a session. The `:Session` calls `at()` method in `:MessageTransporter` and

passes itself as an argument over. The :MessageTransporter then “apply” the :Message to the :Session. The :Session now “apply” the :Message, the :State (stored in the :Session), and itself to the :Messenger. The :Messenger applies to the :State which does the specific data handling of the :Message.

Only MessageTransporter is concrete in Figure 4.11. The Session is an abstract class. The State and Messenger are interfaces which are purely abstract in Java. Session contains State, and Conduits sideA, sideB, and next (inherited from BaseConduit). Messenger and State follows Singleton pattern. Figure 4.12 shows the composed abstract operational

Figure 4.12 Composed Abstract Operational Component
CompositeSession

component CompositeSession which is obtained by aggregating the AMSC from Fig. 4.11.

In Java or C++ implementation, the CompositeSession is the Session itself. But in COM, it could be an interface method, where the interface is really a wrapper which is binary-level reusable on Microsoft Windows.

Using the algorithms described in Chapter 3, the aggregation of execution times for performance evaluation can be done as follows:

- Average time for State.apply(Message, Messenger): $t_{ave, State.apply}$. It is a function of Messenger and Message (i.e., their concrete implementation). It will also depends on what implementation is done in the concrete State.
- Average time for Messenger.apply(Message, State, Session):

$$t_{ave, Messenger.apply} = t_{ave, State.apply} + t_{ave, Messenger.apply.INTERNAL} \quad (4-1)$$

The $t_{ave, Messenger.apply.INTERNAL}$ is the average time the Messenger spends for the call Messenger.apply(Message, State, Session) internally. This is introduced because the MSC or AMSC at this design level has no interests on other calls other than the method call. It is possible to use more detailed MSC if the user found that this item is significant compared with $t_{ave, State.apply}$ and the user wants to improve it.

$t_{ave, Messenger.apply}$ is a function of the concrete implementations of State, Session, and the Messenger itself, and possibly in most of situations, Message also (the Message is neither an abstract class nor a final class; different Message may be designed for different applications).

- Average time for Session.apply(Message):

$$t_{ave,Session.apply} = t_{ave,Messenger.apply} + t_{ave,Session.apply.INTERNAL} \quad (4-2)$$

The $t_{ave,Session.apply.INTERNAL}$ is the average time the Session spends for the call `Session.apply(Message)` internally.

$t_{ave,Session.apply}$ is a function of the concrete implementations of State, Messenger, and Session, and Message.

- Average time for `MessageTransporter.at(Conduit)`:

$$t_{ave,MessageTransporter.at} = t_{ave,Session.apply} + t_{ave,MessageTransporter.at.INTERNAL} \quad (4-3)$$

The $t_{ave,MessageTransporter.at.INTERNAL}$ is the average time the MessageTransporter spends for the call `MessageTransporter.at(Conduit)` internally.

$t_{ave,MessageTransporter.at}$ is a function of the concrete implementations of State, Messenger, and Session, and Message.

- Average time for `Session.accept(Visitor)`:

$$t_{ave,Session.accept} = t_{ave,MessageTransporter.at} + t_{ave,Session.accept.INTERNAL} \quad (4-4)$$

The $t_{ave,Session.accept.INTERNAL}$ is the average time the Session spends for the call `Session.accept(Visitor)` internally.

$t_{ave,Session.accept}$ is a function of the concrete implementations of State, Messenger, and Session, and Message.

The maximum, and minimum time calculations are similar. E.g.,

- Maximum time for `State.apply(Message, Messenger)`: $t_{\max, \text{State.apply}}$. It is a function of Messenger and Message (i.e., their concrete implementation). It will also depends on what implementation is done in the concrete State.
- Maximum time for `Messenger.apply(Message, State, Session)`:

$$t_{\max, \text{Messenger.apply}} = t_{\max, \text{State.apply}} + t_{\max, \text{Messenger.apply.INTERNAL}}$$

(4-5)

The $t_{\text{ave, Messenger.apply.INTERNAL}}$ is the average time the Messenger spends for the call `Messenger.apply(Message, State, Session)` internally.

The variance computation is also similar. E.g.,

- Variance of time for `State.apply(Message, Messenger)`: $V_{\text{State.apply}}$. It is a function of Messenger and Message (i.e., their concrete implementation). It will also depends on what implementation is done in the concrete State.
- Variance of time for `Messenger.apply(Message, State, Session)`:

$$V_{\text{Messenger.apply}} = V_{\text{State.apply}} + V_{\text{Messenger.apply.INTERNAL}} \quad (4-6)$$

The $V_{\text{Messenger.apply.INTERNAL}}$ is the variance of time the Messenger spends for the call `Messenger.apply(Message, State, Session)` internally.

The above description has shown that the average, maximum, minimum, and variance of time for the framework which is described in an AMSC depends on the actual concrete implementations. They are functions of what concrete implementations will be used in a specific design. Using a framework for a real problem as we did in the previous section means using the concrete implementation of that framework. We will see that after a

concrete design is done, all these values becomes concrete. We can also see that there are different level of AMSC in our design.

4.2.1 CodeSession

Figure 4.13 shows an Abstract Message Sequence Chart of Java Conduit for CodeSession.

Figure 4.13 The AMSC for CodeSession

Only AppMessenger is abstract in Figure 4.13, which is more concrete than Figure 4.11.

This design provides another layer of abstraction which is flexible for a designer and

saves efforts in design and coding. Figure 4.14 shows the composed abstract operational component CompositeCodeSession obtained by aggregating the AMSC from Fig. 4.13.

Figure 4.14 Composed Abstract Operational Component CompositeCodeSession

The performance information calculation is similar to that for `CompositeSession`. The formulas are generic and the values depend on the implementations of `AppMessenger` to network and from network.

4.2.1.1 CodeSession to network

Figure 4.15 shows the concrete Message Sequence Chart of Java Conduit for `CodeSession` to network.

Figure 4.15 CodeSession to network

Everything is concrete in Figure 4.15. The abstract AppMessenger in Figure 4.13 is replaced by the concrete CodeMessengerToNet. Figure 4.16 shows the composed

Figure 4.16 Composed Concrete Operational Component
CompositeCodeSessionToNet

concrete operational component CompositeCodeSessionToNet.

Using the algorithms described in Chapter 3, the performance information is obtained in the followings:

- Average time for `CodeState.applyToNet(Message, Messenger)`: $t_{ave, State.apply}$. It is concrete.
- Average time for `CodeMessengerToNet.apply(Message, State, Session)`:

$$t_{ave, CodeMessengerToNet.apply} = t_{ave, CodeState.applyToNet} + t_{ave, CodeMessengerToNet.apply.INTERNAL}$$

(4-7)

The $t_{ave, CodeMessengerToNet.apply.INTERNAL}$ is the average time the `CodeMessengerToNet` spends for the call `CodeMessengerToNet.apply(Message, State, Session)` internally.

$t_{ave, CodeMessengerToNet.apply}$ is concrete.

- Average time for `CodeSession.apply(Message)`:

$$t_{ave, CodeSession.apply} = t_{ave, CodeMessengerToNet.apply} + t_{ave, CodeSession.apply.INTERNAL}$$

(4-8)

The $t_{ave, CodeSession.apply.INTERNAL}$ is the average time the `CodeSession` spends for the call `Session.apply(Message)` internally.

$t_{ave, CodeSession.apply}$ is concrete.

- Average time for `MessageTransporter.at(Conduit)`:

$$t_{ave,MessageTransporter.at} = t_{ave,CodeSession.apply} + t_{ave,MessageTransporter.at.INTERNAL}$$

(4-9)

The $t_{ave,MessageTransporter.at.INTERNAL}$ is the average time the MessageTransporter spends for the call MessageTransporter.at(Conduit) internally.

$t_{ave,MessageTransporter.at}$ is concrete.

- Average time for CodeSession.accept(Visitor):

$$\begin{aligned} t_{ave,CompositeCodeSessionToNet.accept} &= t_{ave,CodeSession.accept} \\ &= t_{ave,MessageTransporter.at} + t_{ave,CodeSession.accept.INTERNAL} \end{aligned} \quad (4-10)$$

The $t_{ave,Session.accept.INTERNAL}$ is the average time the Session spends for the call CodeSession.accept(Visitor) internally.

$t_{ave,CodeSession.accept}$ is concrete.

The maximum, and minimum time calculations are similar. E.g.,

- The maximum time for CodeState.applyToNet(Message, Messenger):

$t_{max,CodeState.apply}$. It is concrete.

- The Maximum time for CodeMessengerToNet.apply(Message, State, Session):

$$t_{max,CodeMessengerToNet.apply} = t_{max,CodeState.apply} + t_{max,CodeMessengerToNet.apply.INTERNAL} \quad (4-11)$$

The $t_{max,CodeMessengerToNet.apply.INTERNAL}$ is the maximum time the Messenger spends for the call CodeMessengerToNet.apply(Message, State, Session) internally.

The variance computation is also similar. E.g.,

- The variance of time for `CodeState.applyToNet(Message, Messenger)`:

$V_{\text{CodeState.applyToNet}}$ It is concrete.

- The variance of time for `CodeMessengerToNet.apply(Message, State, Session)`:

$$V_{\text{CodeMessengerToNet.apply}} = V_{\text{CodeState.apply}} + V_{\text{CodeMessengerToNet.apply.INTERNAL}}$$

(4-12)

The $V_{\text{CodeMessengerToNet.apply.INTERNAL}}$ is the variance of time the `CodeMessengerToNet` spends for the call `CodeMessengerToNet.apply(Message, State, Session)` internally.

The above description has shown that the average, maximum, minimum, and variance of time for an application with concrete implementation of the framework are concrete.

4.2.1.2 CodeSession from network

Figure 4.17 shows the concrete Message Sequence Chart of Java Conduit for `CodeSession from network`.

Figure 4.17 CodeSession from network

Everything is concrete in Figure 4.17. The abstract AppMessenger in Figure 4.13 is replaced by the concrete CodeMessengerFromNet. Figure 4.18 shows the composed

Figure 4.18 Composed Concrete Execution Operational CompositeCodeSessionFromNet

concrete operational component CompositeCodeSessionToNet.

The calculation of the average, maximum, minimum, and variance of time is similar to the CompositeCodeSessionToNet.

4.2.2 StructSession

Figure 4.19 shows an Abstract Message Sequence Chart of Java Conduit for StructSession.

Figure 4.19 The AMSC for StructSession

Only AppMessenger is abstract in Figure 4.19. Figure 4.20 shows the composed abstract operational component CompositeStructSession.

Figure 4.20 Composed Abstract Operational Component CompositeStructSession

The performance information calculation is similar to that for CompositeStructSession. The formulas are generic and the values depend on the implementations of AppMessenger to network and from network.

4.2.2.1 StructSession to network

Figure 4.21 shows the concrete Message Sequence Chart of Java Conduit for StructSession to network.

Figure 4.21 StructSession to network

Everything is concrete in Figure 4.21. Figure 4.22 shows the composed concrete operational component `CompositeStructSessionToNet`.

Figure 4.22 Composed Concrete Operational Component `CompositeStructSessionToNet`

The performance analysis is similar to the operational component CompositeCodeSessionToNet.

4.2.2.2 StructSession from network

Figure 4.23 shows the concrete Message Sequence Chart of Java Conduit for StructSession from network.

Figure 4.23 StructSession from network

Everything is concrete in Figure 4.23. Figure 4.24 shows the composed concrete operational component CompositeStructSessionToNet.

Figure 4.24 Composed Concrete Operational Component CompositeStructSessionFromNet

The calculation of the average, maximum, minimum, and variance of time is similar to the CompositeCodeSessionToNet.

4.2.3 Discussions about Session

A session is a type of Conduit for network protocols. It supports the flow of information in two directions: one goes out to the network, the other comes from the network.

The AMSC described in Figures 4.11 or its composed Operational component CompositeSession in Figure 4.12 is abstract. Its performance depends on its concrete implementations; in another words, its performance depends on the parameters which are the concrete classes and the inputs to the Operational component.

There are different level of AMSC and Abstract Operational component in the design. For example, the AMSC CompositeCodeSession for CodeSession is abstract, but is more concrete than the Abstract Operational component CompositeSession described in Figure 4.11. Performance analysis can be conducted at different levels, each having different number of parameters in the functions. Because these functions are based on different implementations of the framework, concrete values are not available unless performance data is available from previous similar implementations. They are useful for the estimation of an implementation for a similar applications.

Performance analysis can be done for Concrete MSCs directly. In a Concrete MSC, all the polymorphic choices have been made and the performance behaviour is settled.

4.3 Design of Operational Components: Protocols

Figure 4.9 shows the AppProtocol Conduit. Figure 4.5 shows the complete picture of application messages flowing in two directions for AppConduit.

Figure 4.25 shows the exact implementation of MessageTransporter from the Jacob2 framework.

Figure 4.25 Run in MessageTransporter

The variable n in Figure 4.25 is the number of Conduits connected together. As Conduit is an interface in Java, its runtime behaviour depends on the implementation of the interface. Due to this reason, in the following discussion of AppProtocol for performance analysis, we will consider it as a sequence of n operations to n conduits.

4.3.1 Design of Operational Components

4.3.1.1 Concrete Operational components

The average time for the operational component in 4.26 is calculated by:

$$\begin{aligned}
 t_{ave, MessageTransporter.run} &= t_{ave, MessageTransporter.run.INTERNAL} + 2t_{ave, AppProtocol.accept} \\
 &+ t_{ave, CompositeCodeSessionFromNet.accept} + t_{ave, CompositeStructSessionFromNet.accept} \quad (4-13)
 \end{aligned}$$

The variance for the MSC:

$$\begin{aligned}
 V_{MessageTransporter.run} &= V_{MessageTransporter.run.INTERNAL} + 2V_{AppProtocol.accept} \\
 &+ V_{CompositeCodeSessionFromNet.accept} + V_{CompositeStructSessionFromNet.accept} \quad (4-14)
 \end{aligned}$$

The maximum and minimum times are similar to Eq. (4-13).

The average time for the operational component in 4.27 is calculated by:

$$\begin{aligned}
 t_{\text{ave,MessageTransporter.run}} &= t_{\text{ave,MessageTransporter.run.INTERNAL}} + 2t_{\text{ave,AppProtocol.accept}} \\
 &+ t_{\text{ave,CompositeCodeSessionToNet.accept}} + t_{\text{ave,CompositeStructSessionToNet.accept}} \quad (4-15)
 \end{aligned}$$

The variance for the MSC:

$$\begin{aligned}
 V_{\text{MessageTransporter.run}} &= V_{\text{MessageTransporter.run.INTERNAL}} + 2V_{\text{AppProtocol.accept}} \\
 &+ V_{\text{CompositeCodeSessionToNet.accept}} + V_{\text{CompositeStructSessionToNet.accept}} \quad (4-16)
 \end{aligned}$$

The maximum and minimum times are similar to Eq. (4-15).

We have illustrated that by using the concept of Operational component, that they capture complete information about behaviour of all concrete objects involved in a given use-case. For example, Eq. (4-13) is different from Eq. (4-15), and Eq.(4-14) is different from Eq. (4-16) even though the corresponding MSC from Fig. 4.26 and Fig. 4.27 respectively, are very similar and involve exactly the same classes and operations. In section 4.3.2, we

will show that traditional MSCs without operational component concept can not describe performance behaviour as clear as.

Discussion: The introduction of Operational component as a concrete concept in design helps to describe the real runtime behaviour. In an object-oriented design, especial when patterns and frameworks are used , it is difficult to use normal MSC for performance analysis. The use of Operational component, together with Abstract MSC and Abstract Operational components, can help to obtain generic or concrete formulas for the aggregation of execution times directly from the MSCs.

4.3.1.2 Looking at AppProtocol as High Level Operational components

We have shown the AppProtocol as a white box in the previous discussion. From the user's point of view, the AppProtocol can be a black-box. That is, the user does not care what is inside it and what detailed operations are involved when AppProtocol.accept(Visitor) is called. We can abstract Figure 4.26, and 4.27 as a blackbox style operational components named CompositeAppProtocolFromNet (Fig. 4.28a) and CompositeAppProtocolToNet (Fig. 4.28b) respectively. Since they involve the same classes and have similar behaviour, we can abstract them into a single abstract operational component shown in Fig. 4.28c.

Figure 4.28 Composed Operational Components

The average time for the operational component in 4.28(a) is calculated by:

$$\begin{aligned}
 t_{ave, CompositeAppProtocolFromNet} &= t_{ave, MessageTransporter.run} \\
 &= t_{ave, MessageTransporter.run.INTERNAL} + 2t_{ave, AppProtocol.accept} \\
 &\quad + t_{ave, CompositeCodeSessionFromNet.accept} + t_{ave, CompositeStructSessionFromNet.accept}
 \end{aligned}
 \tag{4-17}$$

The variance:

$$\begin{aligned}
 V_{CompositeAppProtocolFromNet} &= V_{MessageTransporter.run} \\
 \text{or } V_{CompositeAppProtocolFromNet} &= V_{MessageTransporter.run.INTERNAL} + 2V_{AppProtocol.accept} \\
 &\quad + V_{CompositeCodeSessionFromNet.accept} + V_{CompositeStructSessionFromNet.accept}
 \end{aligned}
 \tag{4-18}$$

The maximum and minimum times are similar.

The average time for the operational component in 4.28(b) is calculated by:

$$\begin{aligned}
t_{ave, CompositeAppProtocolToNet} &= t_{ave, MessageTransporter.run} \\
&= t_{ave, MessageTransporter.run.INTERNAL} + 2t_{ave, AppProtocol.accept} \\
&+ t_{ave, CompositeCodeSessionToNet.accept} + t_{ave, CompositeStructSessionToNet.accept} \quad (4-19)
\end{aligned}$$

The variance:

$$\begin{aligned}
V_{CompositeAppProtocolToNet} &= V_{MessageTransporter.run} \\
&= V_{MessageTransporter.run.INTERNAL} + 2V_{AppProtocol.accept} \\
&+ V_{CompositeCodeSessionToNet.accept} + V_{CompositeStructSessionToNet.accept} \quad (4-20)
\end{aligned}$$

The maximum and minimum times are determined in a similar way.

The operational components in Fig. 4.28 are highly reusable components because they describe high-level use cases.

The composed operational components `ComposedAppProtocolToNet` and `ComposedAppProtocolFromNet` can be used by other applications like `CompositeStructSessionFromNet`, `CompositeStructSessionToNet`, `CompositeCodeSessionFromNet`, and `CompositeCodeSessionToNet` used in the `AppProtocol`. This composition process can be used recursively to build applications.

Figure 4.28(c) shows the abstraction of the two concrete operational components in Figures 4-28(a) and (b). This level of abstraction may be useful in code implementation or high level usage.

4.3.2 MSC in the Exact Java Implementation without Operational Component Concept

We would like to see what will happen with the performance information if the operational component concept is not used in the design. We will show that by simply building MSC from use-cases based on concrete object interaction, we may miss some important behaviour information.

First we consider the case when receiving information from network. The side B of the :AppProtocol receives the information from network and simply sets the next Conduit as :CodeSession (which is really :CompositeCodeSessionFromNet) and does nothing else. The :CodeSession does decoding and sets the next Conduit as :StructSession (which is really :CompositeStructSessionFromNet). The :StructSession does parsing work and build the data structure which is used in the application. The : StructSession also sets the next conduit as :AppProtocol. The :AppProtocol simply sets the next Conduit as that installed at its side A (it could be an Adaptor, other Session, or null). Figure 4.29 shows the corresponding MSC.

It is interesting to notice that the visitor `:MessageTransporter` knows only that the information arrives at `:AppProtocol` (at side B). After that all the subsequent Conduit connection graph is decided by the `AppProtocol`. The visitor walks through the connection graph blindly.

Second, we consider the case when sending information to network. The side A of the `:AppProtocol` sends the information and simply set the next Conduit as `:StructSession` (which is really `:CompositeStructSessionToNet`) and does nothing else. The `:StructSession` builds the information from the application data structure and sets the next Conduit as `:CodeSession` (which is really `:CompositeCodeSessionToNet`). The `:CodeSession` does encoding work and also sets the next conduit as `:AppProtocol`. The `:AppProtocol` simply sets the next Conduit as that installed at its side B (it could be a UDP Adaptor, TCP Adaptor, other Session, or null). Figure 4.30 shows the corresponding MSC.

We can easily see that the average time for both the MSCs in 4.29 and 4.30 is calculated by:

$$\begin{aligned}
 t_{ave, MessageTransporter.run} = & t_{ave, MessageTransporter.run.INTERNAL} + 2t_{ave, AppProtocol.accept} \\
 & + t_{ave, CodeSession.accept} + t_{ave, StructSession.accept} \quad (4-21)
 \end{aligned}$$

The variance for both MSCs:

$$\begin{aligned}
 V_{MessageTransporter.run} = & V_{MessageTransporter.run.INTERNAL} + 2V_{AppProtocol.accept} \\
 & + V_{CodeSession.accept} + V_{StructSession.accept} \quad (4-22)
 \end{aligned}$$

Even though the formulas for the two MSCs are the same, they fail to capture the fact that the operations accept of both :StructSession and :CodeSession have very different behaviour depending on the parameter passed on them. These operations delegate most of their behaviour to different objects based on the parameter value. This information is

properly captured by the analysis based on the Operational Component concept presented in section 4.3.1.1.

4.4 Summary and Discussion

With the implementation of the Application Protocol, the dynamic behaviour of object oriented design in the design and performance analysis have been illustrated.

Different levels of Abstract Message Sequence Chart and Abstract Operational Component have been illustrated with different Session Operational Components.

It has been found that the runtime behaviour of Conduit framework can be well described by the Operational Components. Traditional MSC cannot provide the necessary performance information without the concept of operational component. In literature, it is commonly agreed that it is very hard to do performance analysis for object oriented design due to its dynamic runtime behaviour. The introduction of operational component concept as a concrete design artifact makes it possible to analyze the performance of design patterns and framework directly based on MSCs which describe the operational components. In Chapter 5, we will show that the concrete operational components can be used as test cases.

Discussion: The procedure to create different levels of Operational components from simple to more and more comprehensive via composition described in this Chapter is a proper method to create reusable Operational components (e.g., Protocol pattern). The Operational Components in a design, can be grouped into static software components, as shown in the example. The bottom-up approach is useful to build more and more high-

level operational components when an OO framework or 3rd-party class library/components are available.

It is also possible to adopt a top-down use-case design approach by decomposition of high-level operational component into lower-level operational components recursively, until all the operational components are basic operational components. The case study shows that the top-down decomposition approach has been successfully used to build the application

Implementation experience: The implementation of the operational components designed in this Chapter is done in VisualAge Java. It was found that Smalltalk-style compilation for each “save” in VisualAge speeds up the development, and the debugging tool in VisualAge is powerful to trace the execution path at different levels according to the user’s requirement.

Chapter 5 Performance Measurement

5.1 Introduction

Performance measurements were done for the implementation of the application protocol described in Chapter 4, by using the measurement tool Jprobe on Windows NT.

According to the material provided by KL GROUP (the vender of Jprobe), we found out that, “the Jprobe Java Virtual Machine has been licensed from JavaSoft and has been specially instrumented by KL Group to allow Jprobe to gather the most complete accurate information possible” [KL Group, 1998]. KL Group claims that it is better than other available tool named “java-prof” [KL Group, 1998] in accuracy, completeness and productivity. The major benefit is that Jprobe can be used to measure CPU time on Windows NT (but not on Windows 95) while java-prof measures only elapsed time.

5.2 Preparation for Measurements

Testing is arranged for the performance information for the Application Protocol implemented in this research with the Java Conduit Framework.

5.2.1 Problems to Use Jprobe for Framework Measurement

In the measurement process, we found that the Jprobe does not support well inheritance and polymorphism which are much used in frameworks and design patterns. It can not differentiate between the times used for different subclass implementations of frameworks and patterns.

For the operational components `ComposedCodeSessionToNet`, `ComposedCodeSessionFromNet`, `CompositeStructSessionFromNet`, and `CompositeStructSessionToNet` described in Chapter 4, the exact execution of `accept(:Visitor)` is a final method in `BaseConduit`. Figure 5.1 shows the exact execution path for the first two operations (`accept` and `at`) in `CompositeCodeSession` operational component when we look at it as a whitebox. It applies to operational components `CompositeCodeSessionToNet` and `CompositeCodeSessionFromNet`.

Figure 5.1 Exact Execution Path between *accept* and *at* in CompositeCodeSession

Jprobe executes the program in the exact order as described in Figure 5.1. The problem with Jprobe is that it collects the time measurement for each class.method without analysis. E.g., the time CodeSession.accept is credited for BaseConduit.accept. This is OK if there is only one Conduit CodeSession, and the operation accept is called only once. When there are several Conduits connected together as a network protocol stack, the time for accept operation of all the concrete implementations (e.g., ComposedCodeSessionToNet, ComposedCodeSessionFromNet, CompositeStructSessionFromNet, and CompositeStructSessionToNet) is credited for BaseConduit.accept as a sum. Jprobe provides no way to tell the user who contributes how much to the sum. Probably Jprobe does not think in the way frameworks and patterns work. In reality, the time for different call of accept to different operational component may be quite different. We will show that for ComposedCodeSessionToNet.accept(:Visitor),

ComposedCodeSessionFromNet.accept(:Visitor),

ComposedStructSessionToNet.accept(:Visitor),

ComposedStructSessionToNet.accept(:Visitor) in the next section where the testing results will be analyzed.

One appropriate solution is to measure time for each **object.operation** instead of **class.operation**. If not possible, it must be able to keep track of the final implementation class (e.g., CodeSession) for a hierarchy and credits all the time used for its operation (which could be coded at any level of the hierarchy) to this final class.

This problem makes the direct measurement of a complete protocol stack (which normally consists of more than one Conduit) meaningless. Special arrangements have to be made to obtain required performance information from measurements.

5.2.2 Arrangement for Framework Measurement and Test Cases

In the measurement, a lot of efforts was spent to separate and differentiate the performance information for each object.operation. Here are the experience obtained in the testing:

- Write one main operation as unit measurement code **for a concrete operational component**, as a concrete operational component is at the level the designer is interested in for performance information. Trying one main operation as

measurement code on an abstract operational component makes the different performance data mixed up and hard to analyze.

- If there is dynamic binding related to the connection of the unit with other units, make sure that the unit testing gets the right dynamic binding (because the connection condition is always false). For the Application Protocol implemented with Java Conduit Framework, if there is an alternation for the connection between Conduits, make sure that the unit testing code (without connection with other Conduit) still allow, to get and set the appropriate Messenger and State.
- Remove the Java keyword *final* for main operations in the framework, and copy the code to each concrete implementation class. This helps to differentiate the time used for each concrete object. This is extremely useful for AppProtocol. However, changing the code is very error prone and time consuming.

The measurement cases are arranged according to the concrete operational components described in `ComposedAppProtocolToNet`, and `ComposedAppProtocolFromNet`. There was a main function written for each of them.

(1) Test code for **ComposedCodeSessionToNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    CodeSession codeSess = new CodeSession("CodeSession:codeSess");  
  
    codeSess.sendTest(msg);  
  
    return;  
}
```

```
}
```

(2) Test code for **ComposedCodeSessionFromNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    CodeSession codeSess = new CodeSession("CodeSession:codeSess");  
  
    codeSess.receiveTest(msg);  
  
    return;  
  
}
```

(3) Test code for **ComposedStructSessionFromNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    StructSession structSess = new StructSession("StructSession:structSess");  
  
    structSess.receiveTest(msg);  
  
    return;  
  
}
```

(4) Test code for **ComposedStructSessionToNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    StructSession structSess = new StructSession("StructSession:structSess");  
  
    structSess.sendTest(msg);  
  
}
```

```
    return;  
  
}
```

(5) Test code for **ComposedAppProtocolFromNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    StructSession structSes = new StructSession("structSes");  
  
    CodeSession codeSes = new CodeSession("codeSes");  
  
    AppProtocol prot = new AppProtocol(structSes, codeSes);  
  
    prot.receiveTest(msg);  
  
    return;  
  
}
```

(6) Test code for **ComposedAppProtocolToNet**:

```
public static void main(String args[]) {  
  
    String msg = "Testing message:Hello!";  
  
    StructSession structSes = new StructSession("structSes");  
  
    CodeSession codeSes = new CodeSession("codeSes");  
  
    AppProtocol prot = new AppProtocol(structSes, codeSes);  
  
    prot.sendTest(msg);  
  
    return;  
  
}
```

Each test case corresponds to exactly an operational component described in Chapter 4. This means that if the design is based on operational component concept, the performance analysis, and test case can be conducted with the design directly.

The testing has been done on Windows NT on the DELL PII-333 machine with memory 128 MB. The results are described in the next section.

5.3 Measurement Results

5.3.1 CompositeCodeSessionFromNet

The measurements are done with repetition of 30 using Jprobe. The result is kept in Table 5.1. The analysis is also done using Microsoft Excel 5.0. The results shows that:

- The average values for the CompositeCodeSessionFromNet.accept, MessageTransporter.at, CodeSession.apply. CodeMessengerFromNet.apply, and CodeState.apply are 251.83 ms, 236.05 ms, 236.05 ms, 236.05 ms, and 229.80 ms respectively.
- The 95% intervals for the CompositeCodeSessionFromNet.accept, MessageTransporter.at, CodeSession.apply. CodeMessengerFromNet.apply, and CodeState.apply are 246.53-257.12, 231.42-240.67, 231.42-240.67, 231.42-240.67, and 226.10-233.49 ms respectively.

Table 5.1 Performance Measurement for CompositeCodeSessionFromNet (unit: millisecond)

	Composite	Message		Code	
Repetition	CodeSessionFromNet	Transporter	CodeSession	MessengerFromNet	CodeState
	.accept	.at	.apply	.apply	.apply
1	256.43	236.70	236.70	236.70	236.70
2	246.57	216.98	216.98	216.98	216.98
3	286.02	246.57	246.57	246.57	236.70
4	256.43	236.70	236.70	236.70	236.70
5	266.29	246.57	246.57	246.57	236.70
6	266.29	256.43	256.43	256.43	236.70
7	256.43	246.57	246.57	246.57	236.70
8	236.70	216.98	216.98	216.98	216.98
9	246.57	236.70	236.70	236.70	226.84
10	256.43	246.57	246.57	246.57	236.70
11	256.43	216.98	216.98	216.98	216.98
12	256.43	236.70	236.70	236.70	226.84
13	236.70	226.84	226.84	226.84	216.98
14	226.84	226.84	226.84	226.84	216.98
15	256.43	226.84	226.84	226.84	226.84
16	236.70	226.84	226.84	226.84	226.84
17	256.43	246.57	246.57	246.57	236.70
18	276.15	246.57	246.57	246.57	236.70
19	276.15	236.70	236.70	236.70	236.70
20	246.57	226.84	226.84	226.84	216.98
21	246.57	226.84	226.84	226.84	216.98
22	226.84	216.98	216.98	216.98	216.98
23	256.43	256.43	256.43	256.43	246.57
24	236.70	236.70	236.70	236.70	236.70
25	256.43	236.70	236.70	236.70	226.84
26	236.70	236.70	236.70	236.70	236.70
27	266.29	266.29	266.29	266.29	256.43
28	246.57	246.57	246.57	246.57	236.70
29	226.84	216.98	216.98	216.98	216.98
30	256.43	236.70	236.70	236.70	226.84
Average	251.83	236.05	236.05	236.05	229.80
Stdev	14.79	12.93	12.93	12.93	10.33
95% confidence max	257.12	240.67	240.67	240.67	233.49
95% confidence min	246.53	231.42	231.42	231.42	226.10

5.3.2 CompositeStructFromNet

The testing is done with repetition of 30 using Jprobe. The results are given in Table 5.2.

The analysis is also done using Microsoft Excel 5.0. The results show that:

- The average values for the CompositeStructSessionFromNet.accept, MessageTransporter.at, StructSession.apply. StructMessengerFromNet.apply, and StructState.apply are 54.57 ms, 37.48 ms, 37.48 ms, 37.15 ms, and 31.56 ms respectively.

The 95% intervals for the CompositeStructSessionFromNet.accept, MessageTransporter.at, StructSession.apply. StructMessengerFromNet.apply, and StructState.apply are 51.41-57.73, 39.87-35.09, , 39.87-35.09, 34.90-39.40, and 29.28-33.84 ms respectively.

Table 5.2 Performance Measurement for CompositeStructSessionFromNet (unit: millisecond)

	Composite	Message		Struct	
Repetition	StructSessionFromNet	Transporter	StructSession	MessengerFromNet	StructState
	.accept	.at	.apply	.apply	.apply
1	59.18	49.31	49.31	49.31	39.45
2	69.04	49.31	49.31	49.31	39.45
3	59.18	49.31	49.31	49.31	39.45
4	39.45	29.59	29.59	29.59	29.59
5	59.18	29.59	29.59	29.59	29.59
6	59.18	39.45	39.45	39.45	29.59
7	49.31	39.45	39.45	39.45	29.59
8	49.31	39.45	39.45	39.45	39.45
9	59.18	39.45	39.45	39.45	29.59
10	59.18	39.45	39.45	39.45	39.45
11	49.31	29.59	29.59	29.59	19.73
12	69.04	49.31	49.31	39.45	39.45
13	49.31	29.59	29.59	29.59	19.73
14	69.04	39.45	39.45	39.45	29.59
15	59.18	39.45	39.45	39.45	29.59
16	49.31	39.45	39.45	39.45	39.45
17	49.31	29.59	29.59	29.59	19.73
18	59.18	39.45	39.45	39.45	29.59
19	59.18	29.59	29.59	29.59	29.59
20	59.18	39.45	39.45	39.45	29.59
21	49.31	29.59	29.59	29.59	29.59
22	39.45	29.59	29.59	29.59	29.59
23	39.45	29.59	29.59	29.59	29.59
24	49.31	29.59	29.59	29.59	29.59
25	59.18	39.45	39.45	39.45	19.73
26	49.31	39.45	39.45	39.45	29.59
27	69.04	39.45	39.45	39.45	39.45
28	59.18	39.45	39.45	39.45	39.45
29	39.45	29.59	29.59	29.59	29.59
30	49.31	49.31	49.31	49.31	39.45
Average	54.57	37.48	37.48	37.15	31.56
Stddev	8.83	6.68	6.68	6.29	6.38
95% confidence max	57.73	39.87	39.87	39.40	33.84
95% confidence min	51.41	35.09	35.09	34.90	29.28

5.3.3 CompositeAppProtocolFromNet

The testing is done with repetition of 30. The result is kept in Table 5.3. The analysis is done using Microsoft Excel 5.0. The results shows that:

- The average values for the CompositeCodeSessionFromNet.accept, Composite StructSessionFromNet.accept, AppProtocol.accept, and CompositeAppProtocolFromNet.accept are 252.48 ms, 53.92 ms, 0.00ms, and 306.40 ms respectively. For CompositeCodeSessionFromNet.accept, the value obtained in its unit test is 251.83 with difference only 0.65 ms or 0.26%. For Composite StructSessionFromNet.accept, , the value obtained in its unit test is 54.57 with difference only 0.65 ms or 1.19%. These are considered as reasonable error (5% error is considered as acceptable). We also notice that both average values are with the 95% confidence range of the other measurement. In the data collection, the result from the unit tests are used.
- The 95% intervals for CompositeCodeSessionFromNet.accept, Composite StructSessionFromNet.accept, AppProtocol.accept, and CompositeAppProtocolFromNet.accept are 247.90-257.07, 51.20-56.63, 0.00-0.00, 301.63-311.17 respectively.
- It is noticed that the time used for the 2 AppProtocol.accept operations (see Figure 4.24) has the average 0.00 ms. This means that the Protocol pattern costs very

Table 5.3 Performance Measurement for CompositeAppProtocolFromNet (unit: millisecond)

	Composite	Composite		Composite
Repetition	CodeSessionFrom Net	StructSessionFrom Net	AppProtoc ol	AppProtocolFro mNet
	.accept	.accept	.accept	.accept.
1	256.43	59.18	0.00	315.61
2	276.15	49.31	0.00	325.46
3	266.29	49.31	0.00	315.60
4	256.43	39.45	0.00	295.88
5	246.57	59.18	0.00	305.75
6	246.57	59.18	0.00	305.75
7	226.84	49.31	0.00	276.15
8	266.29	49.31	0.00	315.60
9	256.43	39.45	0.00	295.88
10	226.84	49.31	0.00	276.15
11	256.43	59.18	0.00	315.61
12	236.70	59.18	0.00	295.88
13	256.43	49.31	0.00	305.74
14	246.57	59.18	0.00	305.75
15	266.29	49.31	0.00	315.60
16	256.43	49.31	0.00	305.74
17	226.84	59.18	0.00	286.02
18	256.43	59.18	0.00	315.61
19	246.57	49.31	0.00	295.88
20	276.15	49.31	0.00	325.46
21	256.43	69.04	0.00	325.47
22	256.43	49.31	0.00	305.74
23	246.57	59.18	0.00	305.75
24	266.29	59.18	0.00	325.47
25	256.43	49.31	0.00	305.74
26	236.70	59.18	0.00	295.88
27	246.57	59.18	0.00	305.75
28	246.57	59.18	0.00	305.75
29	256.43	69.04	0.00	325.47
30	256.43	39.45	0.00	295.88
Average	252.48	53.92	0.00	306.40
Stddev	12.82	7.59	0.00	13.33
95% confidence	257.07	56.63	0.00	311.17
max				
95% confidence	247.90	51.20	0.00	301.63
min				

little in the AppProtocol for the application protocol implemented in Chapter 4. The overhead of Protocol pattern in this design is negligible.

5.3.4 CompositeStructSessionToNet

The testing is done with repetition of 30 using Jprobe. The result is kept in Table 5.4.

The analysis is also done using Microsoft Excel 5.0. The results shows that:

- The average values for the CompositeStructSessionToNet.accept, MessageTransporter.at, StructSession.apply. StructMessengerToNet.apply, and StructState.apply are 41.75 ms, 18.09 ms, 18.09 ms, 17.43 ms, and 10.85 ms respectively.
- The 95% intervals for the CompositeStructSessionToNet.accept, MessageTransporter.at, StructSession.apply. StructMessengerToNet.apply, and StructState.apply are 38.96-44.54, 16.75-19.42, , 16.75-19.42, 15.92-18.94, and 9.77-11.92 ms respectively.

Table 5.4 Performance Measurement for CompositeStructSessionToNet (unit: millisecond)

	Composite	Message		Struct	
Repetition	StructSessionToNet	Transporter	StructSession	MessengerToNet	StructState
	.accept	.at	.apply	.apply	.apply
1	59.18	19.73	19.73	19.73	9.86
2	29.59	9.86	9.86	9.86	9.86
3	39.45	19.73	19.73	19.73	9.86
4	29.59	9.86	9.86	9.86	9.86
5	29.59	19.73	19.73	19.73	9.86
6	39.45	19.73	19.73	19.73	9.86
7	39.45	19.73	19.73	19.73	19.73
8	39.45	19.73	19.73	19.73	9.86
9	49.31	19.73	19.73	19.73	19.73
10	29.59	9.86	9.86	9.86	9.86
11	29.59	9.86	9.86	9.86	9.86
12	49.31	19.73	19.73	19.73	19.73
13	49.31	19.73	19.73	19.73	9.86
14	49.31	19.73	19.73	19.73	9.86
15	49.31	19.73	19.73	19.73	9.86
16	39.45	19.73	19.73	19.73	9.86
17	39.45	9.86	9.86	9.86	9.86
18	59.18	19.73	19.73	19.73	9.86
19	39.45	19.73	19.73	19.73	9.86
20	39.45	19.73	19.73	19.73	9.86
21	49.31	19.73	19.73	19.73	9.86
22	49.31	19.73	19.73	19.73	9.86
23	29.59	19.73	19.73	19.73	9.86
24	49.31	19.73	19.73	19.73	9.86
25	39.45	19.73	19.73	19.73	9.86
26	39.45	19.73	19.73	19.73	9.86
27	39.45	19.73	19.73	9.86	9.86
28	39.45	19.73	19.73	9.86	9.86
29	39.45	19.73	19.73	19.73	9.86
30	49.31	19.73	19.73	19.73	9.86
Average	41.75	18.09	18.09	17.43	10.85
Stdev	7.80	3.73	3.73	4.22	3.01
95% confidence max	44.54	19.42	19.42	18.94	11.92
95% confidence min	38.96	16.75	16.75	15.92	9.77

5.3.5 CompositeCodeSessionToNet

Table 5.5 Performance Measurement for CompositeCodeSessionToNet (unit: millisecond)

	Composite	Message		Code	
Repetition	CodeSessi onToNet	Transport er	CodeSessi on	Messeng erToNet	CodeStat e
	.accept	.accept	.apply	.apply	.apply
1	147.94	138.08	138.08	138.08	128.21
2	138.08	118.35	118.35	118.35	108.49
3	138.08	138.08	138.08	138.08	128.21
4	138.08	118.35	118.35	118.35	118.35
5	147.94	138.08	138.08	138.08	118.35
6	147.94	128.21	128.21	128.21	108.49
7	128.21	108.49	108.49	108.49	108.49
8	128.21	118.35	118.35	118.35	118.35
9	128.21	128.21	128.21	128.21	128.21
10	147.94	118.35	118.35	118.35	108.49
11	138.08	118.35	118.35	118.35	118.35
12	157.80	138.08	138.08	128.21	118.35
13	138.08	128.21	128.21	128.21	118.35
14	138.08	138.08	138.08	138.08	128.21
15	138.08	118.35	118.35	118.35	108.49
16	147.94	128.21	128.21	128.21	118.35
17	138.08	128.21	128.21	128.21	128.21
18	147.94	138.08	138.08	138.08	118.35
19	138.08	128.21	128.21	128.21	118.35
20	138.08	128.21	128.21	128.21	128.21
21	128.21	108.49	108.49	108.49	108.49
22	128.21	128.21	128.21	128.21	128.21
23	138.08	128.21	128.21	128.21	118.35
24	108.49	98.63	98.63	98.63	98.63
25	147.94	128.21	128.21	128.21	118.35
26	138.08	128.21	128.21	128.21	118.35
27	138.08	138.08	138.08	138.08	128.21
28	128.21	118.35	118.35	118.35	118.35
29	138.08	128.21	128.21	128.21	118.35
30	128.21	118.35	118.35	118.35	108.49
Average	137.75	125.58	125.58	125.25	118.02
Stdev	9.31	9.73	9.73	9.44	7.74
95% confidence max	141.08	129.06	129.06	128.63	120.79
95% confidence	134.42	122.10	122.10	121.88	115.25

min					
-----	--	--	--	--	--

The testing is done with repetition of 30 using Jprobe. The result is kept in Table 5.5.

The analysis is also done using Microsoft Excel 5.0. The results shows that:

- The average values for the CompositeCodeSessionToNet.accept, MessageTransporter.at, CodeSession.apply, CodeMessengerToNet.apply, and CodeState.apply are 137.75 ms, 125.58 ms, 125.58 ms, 125.25 ms, and 118.02 ms respectively.
- The 95% intervals for the CompositeCodeSessionToNet.accept, MessageTransporter.at, CodeSession.apply, CodeMessengerToNet.apply, and CodeState.apply are 134.42-141.08, 122.10-129.06, 122.10-129.06, 121.88-128.63, and 115.25-120.79 ms respectively.

5.3.6 CompositeAppProtocolToNet

The testing is done with repetition of 30. The result is kept in Table 5.6. The analysis is done using Microsoft Excel 5.0. The results shows that:

- The average values for the CompositeCodeSessionToNet.accept, Composite StructSessionToNet.accept, AppProtocol.accept, and CompositeAppProtocolToNet.accept are 135.45 ms, 42.74ms, 0.00ms, and 178.19 ms respectively. For CompositeCodeSessionToNet.accept, the value obtained in

its unit test is 137.75 with difference only 2.3 ms or 1.67%. For Composite StructSessionToNet.accept, , the value obtained in its unit test is 41.75 with difference only 0.99 ms or 2.37%. These are considered as reasonable error (if it is less than 5%). We also notice that both average values are with the 95% confidence range of the other measurement. In the data collection, the result from the unit tests are used.

- The 95% intervals for CompositeCodeSessionToNet.accept, Composite StructSessionToNet.accept, AppProtocol.accept, and CompositeAppProtocolToNet.accept are 132.12-138.78, 40.45-45.03, 0.00-0.00, 174.42-181.95 respectively.

Table 5.6 Performance Measurement CompositeAppProtocolToNet (unit: millisecond)

	Composite	Composite		Composite
Repetition	CodeSessionFrom Net	StructSessionFrom Net	AppProtoc ol	AppProtocolFro mNet
	.accept	.accept	.accept	.accept.
1	138.08	49.31	0.00	187.39
2	147.94	49.31	0.00	197.25
3	138.08	39.45	0.00	177.53
4	138.08	39.45	0.00	177.53
5	128.21	49.31	0.00	177.52
6	128.21	49.31	0.00	177.52
7	147.94	49.31	0.00	197.25
8	147.94	39.45	0.00	187.39
9	128.21	39.45	0.00	167.66
10	138.08	39.45	0.00	177.53
11	118.35	39.45	0.00	157.80
12	128.21	39.45	0.00	167.66
13	147.94	49.31	0.00	197.25
14	128.21	49.31	0.00	177.52
15	138.08	39.45	0.00	177.53
16	147.94	29.59	0.00	177.53
17	128.21	39.45	0.00	167.66
18	128.21	39.45	0.00	167.66
19	128.21	29.59	0.00	157.80
20	128.21	49.31	0.00	177.52
21	138.08	49.31	0.00	187.39
22	128.21	39.45	0.00	167.66
23	138.08	39.45	0.00	177.53
24	147.94	39.45	0.00	187.39
25	147.94	39.45	0.00	187.39
26	138.08	39.45	0.00	177.53
27	118.35	49.31	0.00	167.66
28	147.94	39.45	0.00	187.39
29	128.21	39.45	0.00	167.66
30	128.21	59.18	0.00	187.39
Average	135.45	42.74	0.00	178.18
Stdev	9.30	6.40	0.00	10.52
95% confidence	138.78	45.03	0.00	181.95
max				
95% confidence	132.12	40.45	0.00	174.42

min				
-----	--	--	--	--

5.3.7 Summary of the Performance Data for Each Concrete Operational component

$$\text{Variance} = \text{Stdev}^2 \quad (5-1)$$

We have seen that each test case corresponds to exactly an operational component described in Chapter 4. Now we are able to summarize the average time and variance for each operational component in Chapter 4.

(1) CompositeStructSessionToNet

Result in Table 5.4 is used for data collection.

$$t_{\text{ave, StructState.applyToNet}} = 10.85 \text{ ms}; V_{\text{ave, StructState.applyToNet}} = 3.01^2 = 9.06.$$

$$t_{\text{ave, StructMessengerToNet.apply}} = 17.43 \text{ ms}; V_{\text{StructMessengerToNet.apply}} = 4.22^2 = 17.81.$$

$$t_{\text{ave, StructMessengerToNet.apply.INTERNAL}} = t_{\text{ave, StructMessengerToNet.apply}} - t_{\text{ave, StructState.applyToNet}} = 6.58.$$

$V_{\text{StructMessengerToNet.apply.INTERNAL}} = V_{\text{StructMessengerToNet.apply}} - V_{\text{StructState.applyToNet}} = 8.75.$ (If it is minus, it should be adjusted to 0.) The INTERNAL time and variance for other operations can be obtained in the similar calculation.

$$t_{\text{ave, StructSession.apply}} = 18.09 \text{ ms}; V_{\text{StructSession.apply}} = 3.73^2 = 13.91.$$

$$t_{\text{ave, MessageTransporter.at}} = 18.09 \text{ ms}; V_{\text{MessageTransporter.at}} = 13.91.$$

$$t_{\text{ave, CompositeStructSessionToNet.accept}} = t_{\text{ave, StructSession.accept}} = 41.75;$$

$$V_{\text{CompositeStructSessionToNet.accept}} = V_{\text{StructSession.accept}} = \max \{7.80^2, V_{\text{MessageTransporter.at}}\}$$

$$= \max \{60.84, 41.75\} = 60.84.$$

(2) CompositeCodeSessionToNet

Result in Table 5.5 is used for data collection.

$$t_{\text{ave,CodeState.applyToNet}} = 118.02\text{ms}; V_{\text{ave,CodeState.applyToNet}} = 7.74^2 = 59.91.$$

$$t_{\text{ave, CodeMessengerToNet.apply}} = 125.25\text{ms}; V_{\text{CodeMessengerToNet.apply}} = 9.44^2 = 89.11.$$

$$t_{\text{ave, CodeMessengerToNet.apply.INTERNAL}} = t_{\text{ave, CodeMessengerToNet.apply}} - t_{\text{ave,CodeState.applyToNet}} = 7.23.$$

$V_{\text{CodeMessengerToNet.apply.INTERNAL}} = V_{\text{CodeMessengerToNet.apply}} - V_{\text{CodeState.applyToNet}} = 29.20$. (If it is minus, it should be adjusted to 0.) The INTERNAL time and variance for other operations can be obtained in the similar calculation.

$$t_{\text{ave,CodeSession.apply}} = 125.58 \text{ ms}; V_{\text{CodeSession.apply}} = 9.73^2 = 94.67.$$

$$t_{\text{ave,MessageTransporter.at}} = 125.58 \text{ ms}; V_{\text{MessageTransporter.at}} = 94.67.$$

$$t_{\text{ave,CompositeCodeSessionToNet.accept}} = t_{\text{ave,CodeSession.accept}} = 137.75;$$

$$V_{\text{CompositeCodeSessionToNet.accept}} = V_{\text{CodeSession.accept}} = \max \{9.31^2, V_{\text{MessageTransporter.at}}\}$$

$= \max \{86.68, 94.67\} = 94.67$. (This keeps the data in the safe side because the variance for $V_{\text{CodeSession.accept.INTERNAL}}$ should be larger or equal to 0 on the safe side.)

(3) CompositeAppProtocolToNet

Result in Table 5.6 is used for data collection.

$$t_{\text{ave,AppProtocol.accept}} = 0; t_{\text{ave,MessageTransporter.run.INTERNAL}} = 0;$$

$$t_{\text{ave,CompositeAppProtocolToNet}} = t_{\text{ave,MessageTransporter.run.INTERNAL}} + 2t_{\text{ave,AppProtocol.accept}}$$

$$+ t_{ave, CompositeCodeSessionToNet.accept} + t_{ave, CompositeStructSessionToNet.accept} = 178.18$$

ms;

$$V_{CompositeAppProtocolToNet} = 10.52^2 = 110.67.$$

(4) CompositeStructSessionFromNet

Result in Table 5.2 is used for data collection.

$$t_{ave, StructState.applyFromNet} = 31.56 \text{ ms}; V_{ave, StructState.applyFromNet} = 6.38^2 = 40.70.$$

$$t_{ave, StructMessengerFromNet.apply} = 37.15 \text{ ms};$$

$$V_{StructMessengerFromNet.apply} = \max\{6.29^2, V_{ave, StructState.applyFromNet}\} = 40.70.$$

$$t_{ave, StructSession.apply} = 37.48 \text{ ms}; V_{StructSession.apply} = 6.68^2 = 44.62.$$

$$t_{ave, MessageTransporter.at} = 37.48 \text{ ms}; V_{MessageTransporter.at} = 6.68^2 = 44.62.$$

$$t_{ave, CompositeStructSessionFromNet.accept} = t_{ave, StructSession.accept} = 54.57 \text{ ms};$$

$$V_{CompositeStructSessionFromNet.accept} = V_{StructSession.accept} = \max\{8.83^2, V_{MessageTransporter.at}\} \\ = \max\{77.97, 54.57\} = 77.97.$$

(5) CompositeCodeSessionFromNet

Result in Table 5.1 is used for data collection.

$$t_{ave, CodeState.applyFromNet} = 229.80 \text{ ms}; V_{ave, CodeState.applyFromNet} = 10.33^2 = 106.71.$$

$$t_{ave, CodeMessengerFromNet.apply} = 236.05 \text{ ms}; V_{CodeMessengerFromNet.apply} = 12.93^2 = 167.18.$$

$$t_{ave, CodeSession.apply} = 236.05 \text{ ms}; V_{CodeSession.apply} = 12.93^2 = 167.18.$$

$$t_{ave,MessageTransporter.at} = 236.05 \text{ ms}; V_{MessageTransporter.at} = 12.93^2 = 167.18.$$

$$t_{ave,CompositeCodeSessionFromNet.accept} = t_{ave,CodeSession.accept} = 251.83 \text{ ms};$$

$$V_{CompositeCodeSessionFromNet.accept} = V_{CodeSession.accept} = \max \{14.79^2, V_{MessageTransporter.at}\} \\ = \max \{218.74, 167.18\} = 218.74.$$

(6) CompositeAppProtocolFromNet

Result in Table 5.3 is used for data collection.

$$t_{ave,AppProtocol.accept} = 0; t_{ave,MessageTransporter.run.INTERNAL} = 0;$$

$$t_{ave,CompositeAppProtocolFromNet} = t_{ave,MessageTransporter.run.INTERNAL} + 2t_{ave,AppProtocol.accept} \\ + t_{ave,CompositeCodeSessionFromNet.accept} + t_{ave,CompositeStructSessionFromNet.accept} = 306.40 \\ \text{ms};$$

$$V_{CompositeAppProtocolFromNet} = 13.33^2 = 177.69.$$

5.3.8 Summary of the Performance Data for Each Abstract Operational component

With the concept of abstract operational component, it is now possible to provide performance information at an abstract level.

(1) CompositeCodeSession

$$t_{ave,CompositeCodeSession.accept} = \begin{cases} 251.83\text{ms}, \text{AppMessenger} = \text{CodeMessengerFromNet} \\ 137.75\text{ms}, \text{AppMessenger} = \text{CodeMessengerToNet} \end{cases}$$

$$V_{\text{CompositeCodeSession.accept}} = \begin{cases} 218.74 \text{ ms}^2, \text{AppMessenger} = \text{StructMessengerFromNet} \\ 94.67 \text{ ms}^2, \text{AppMessenger} = \text{StructMessengerToNet} \end{cases}$$

(2) CompositeStructSession

$$t_{\text{ave.CompositeStructSession.accept}} = \begin{cases} 54.57 \text{ ms}, \text{AppMessenger} = \text{StructMessengerFromNet} \\ 41.75 \text{ ms}, \text{AppMessenger} = \text{StructMessengerToNet} \end{cases}$$

$$V_{\text{CompositeStructSession.accept}} = \begin{cases} 77.97 \text{ ms}^2, \text{AppMessenger} = \text{StructMessengerFromNet} \\ 60.84 \text{ ms}^2, \text{AppMessenger} = \text{StructMessengerToNet} \end{cases}$$

(3) CompositeSession

$$t_{\text{ave.CompositeSession.accept}} = \begin{cases} 54.57 \text{ ms}, \text{Messenger} = \text{StructMessengerFromNet}, \\ \quad \text{Session} = \text{StructSession}, \text{State} = \text{StructState}; \\ 41.75 \text{ ms}, \text{Messenger} = \text{StructMessengerToNet}, \\ \quad \text{Session} = \text{StructSession}, \text{State} = \text{StructState}; \\ 251.83 \text{ ms}, \text{Messenger} = \text{CodeMessengerFromNet}, \\ \quad \text{Session} = \text{CodeSession}, \text{State} = \text{CodeState}; \\ 137.75 \text{ ms}, \text{Messenger} = \text{CodeMessengerToNet}, \\ \quad \text{Session} = \text{CodeSession}, \text{State} = \text{CodeState} \end{cases}$$

$$V_{\text{CompositeSession.accept}} = \begin{cases} 77.97 \text{ ms}^2, \text{Messenger} = \text{StructMessengerFromNet}, \\ \quad \text{Session} = \text{StructSession}, \text{State} = \text{StructState}; \\ 60.84 \text{ ms}^2, \text{Messenger} = \text{StructMessengerToNet}, \\ \quad \text{Session} = \text{StructSession}, \text{State} = \text{StructState}; \\ 218.74 \text{ ms}^2, \text{Messenger} = \text{CodeMessengerFromNet}, \\ \quad \text{Session} = \text{CodeSession}, \text{State} = \text{CodeState}; \\ 94.67 \text{ ms}^2, \text{Messenger} = \text{CodeMessengerToNet}, \\ \quad \text{Session} = \text{CodeSession}, \text{State} = \text{CodeState}. \end{cases}$$

These measures allow the designer to predict the execution time for new protocols based on the framework, before the implementation phase.

5.4 Summary

Performance measurement is done with Jprobe and the result is analyzed with Microsoft Excel.

Each operational component is a test case in the measurement design. Special arrangements have been done to isolate the CPU time contributed by each interested operation.

Performance data for the concrete and abstract operational components described in Chapter 4 are characterized. For the first time, performance information of abstract operational components is provided for performance analysis.

It has been found that Jprobe is useful tool for the performance measurement for Java applications. However currently it does not fit into the measurement of design patterns and framework. Major design change for time computation is required for Jprobe to measure an OO framework directly without difficulty. Our research analyzed Jprobe, and made suggestions for this design change.

With the problem in Jprobe, we successfully conducted design and code changes in our case study to isolate the polymorphism in the framework before the measurements.

Chapter 6 Conclusions

6.1 Conclusions

The thesis successfully integrate the use case composition in OO design with resource demand aggregation for performance analysis. The operational component concept, and extensions to MSC and performance analysis algorithms are the keys to make this integration happen. The proposed methodology is useful for O-O design and especially for component oriented design.

The results of the research are summarized as the followings:

1. Concepts Operational Component, Abstract Operational Component, Abstract Message Sequence Chart are proposed for object oriented design. These new concepts are used both for building an OO system and for evaluating its performance. They are very useful especially when design patterns and framework are used.
2. Extensions to message sequence chart are provided to include performance information which is necessary for the performance analysis of object oriented design.
3. Rules for use case composition for O-O design are defined. Structural graph reduction rules for execution time aggregation are adopted for the creation of operational component using composition approach in object oriented design.

4. A detailed case study for use case composition of application protocol design with Java Conduit framework is conducted. The dynamic behaviour of the framework from performance point of view is demonstrated. Different level of AMSCs and Abstract Operational Components have been used in the design.

Both decomposition and composition approaches have been used in the design.

Top-down decomposition approach is used for use case driven O-O design for high-level design. Composition approach is used for operational component

design and its performance analysis. Code implementation is done with

VisualAge Java on Windows NT.

The case study using operational component concept has produced the following interesting results:

- (a) Operational components can describe performance correctly for a framework while traditional MSC may miss relevant information;
- (b) Mapping from use cases to operational components is straightforward while mapping from operational components to classes may require both domain knowledge and design experience.

These results shows that using (concrete and abstract) operational component helps to fill the gap between use case design and object orientation.

5. Performance measurement is done for each operational components in the application protocol on Windows NT. Performance information is described at both concrete and abstract levels for performance analysis. Both concrete and abstract operational components become reusable in a design.

6. Implementation experience:

(a) Jprobe is a useful tool for the performance measurement of Java applications.

But the thesis also found that major design change is required for Jprobe to deal properly with inheritance and polymorphism which are frequently used in framework and design patterns. Our research analyzed Jprobe, and made suggestions for this design change.

To overcome this difficulty without changing Jprobe, in the research, design and code changes have been conducted to allow correct time measurement for each concrete operational component in the case study.

(b) VisualAge Java provides a nice developed environment. Its facilities for debugging are very useful.

6.2 Suggestions for Future Research

1. It is suggested to design a CASE tool for design and performance analysis using the operational component concepts, and extensions to MSC and analysis algorithms which are proposed in the thesis. It is also possible to work with some industrial CASE tools to include them (e.g. Rational Rose, ObjectTime, ...).
2. It is suggested to work with some available performance measurement tools (e.g. Jprobe) to allow direct measurement of OO frameworks/design patterns possible.

Appendix 1 Setup of AppProtocol

Let's see the Message Sequence Chart for the setup of AppProtocol implemented in Java.

Figure A1.1 MSC: Setup of AppProtocol

The Conduit `insideA` and `insideB` are `:StructSession` and `:CodeSession` respectively. The result of the MSC in Figure A1.1 is the `AppProtocol` plotted in Figure 4.9. The calls 1.1 and 1.2 in the MSC which connected the two sides are inherited from the super class

Protocol. The connection between insideA and insideB is decided by the real implementation. In AppProtocol, they are simply connected together.

The average time used for the setup is

$$t_{ave,AppProtocol.new} = t_{ave,AppProtocol..new.INTERNAL} + 2t_{ave,..StructSession.attach} + 2t_{ave.CodeSession.attach}$$

(A1-1)

Here $t_{ave,AppProtocol..new.INTERNAL}$ is the time used for the condition check.

The variance:

$$V_{AppProtocol.new} = V_{AppProtocol..new.INTERNAL} + 2V_{StructSession.attach} + 2V_{CodeSession.attach}$$

(A1-2)

The maximum and minimum times are similar to Eq. (A1-1).

References

1. Adam Denning, ActiveX Controls Inside Out, 2nd edition, Microsoft Press, 1997.
2. Anderson Consulting, Component-based software engineering (CBSE) project.
Software Engineering Laboratory, Center for Strategic Technology Research,
Anderson Consulting
(http://www.ac.com/aboutus/tech/cstar/sel/frames/au_dfrsel.html), 1997.
3. Arnold K. and Gosling J., The Java Programming Language, Addison-Wesley,
Reading, MA, 1996.
4. ATM Forum. ATM UNI specification Q.93B. Prentice Hall, 1993.
5. ATM Forum. Private Network-Network Interface Specification Version 1.0
(PNNI 1.0). 1996
6. Beck, K., and Johnson, R.. Patterns generate architectures. Technical report,
University of Illinois at Urbana-Champaign, 1994.
7. Booch, G., Object-Oriented Analysis and Design with Applications, Redwood
City, CA, Benjamin.Cummings, 1994.
8. Buhr, R.J.A., and Casselman, R.S., "Timethread-RoleMaps for Object-Oriented
Design of Real-Time and Distributed Systems, Proceedings of OOPSLA '94:

- Object-Oriented Programming Systems, Languages and Applications, Portland, OR, October, 1994, pp.301-316.
9. Buhr, R.J.A., and Casselman, R.S., "A Use Case Map Approach to High Level Design of Object Oriented Systems", Prentice-Hall, Inc., 1996.
 10. Calder, B, Shannon, B., JavaBeans Activation Framework Specification Version 1.0. Sun Microsystems. March 1998.
 11. Dewire, D. T., Application Development for Distributed Environments, pp. 149-157, McGraw-Hill, Inc., 1994.
 12. Eichner B., Kamber D., and Murer S., CORBA: Principle and practical experiences. Informatik/Informatique, February 1997.
 13. Engei, R., Bieri, T., Keller, B. Signalling in ATM Networks: Experience with an Object-Oriented Solution. ASCOM TECH, Switzerland. 1995.
 14. FAMOOS Consortium, Framework-based Approach for Mastering Object-Oriented Software Evolution, ESPRIT Project 21975, <http://www.sema.es/projects/FAMOOS/>, 1996.
 15. Fowler, M. and Scott K.. UML Distilled: Applying the Standard Object Modelling Language, Addison-Wesley, An imprint of Addison Wesley Longman, Inc., 1997.
 16. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company. 1995.

17. Hrischuk, C., Rolia, J., and Woodside, C. M., “Automatic Generation of a Software Model Using an Object-Oriented Prototype”, Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Durham, NC, January, 1995, pp.399-409.
18. Huni, H., Johnson, R., Engel, R.. A Framework for Network Protocol Software. ACM, 1995.
19. IONA, Orbix Desktop for Windows. White Paper. IONA Technologies, <http://www.iona.com>, July 1996.
20. ITU-T Message Sequence Chart (MSC). Series Z: Programming Languages. Z120. 1996.
21. ITU-T Recommendation Q.934 (Frame Relay)
22. ITU-TS Recommendation Q.2931.
23. Jacobson, I., and etc.. Objectory with UML Syntax. Addison-Wesley Publishing Company. 1997
24. Jacobson, I., and etc.. Use Case Design with UML Syntax. Addison-Wesley Publishing Company. 1997
25. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Publishing Company. 1992.

26. Jain, R., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons, Inc. 1991.
27. Johnson, R.E., How to Design Frameworks, Notes for OOPSLA '93, 1993.
28. Keller, R.K., and Schauer, R.. Design Components: Towards Software Composition at the Design Level. Technical report #1096, Departement IRO, University de Montreal, November 1997.
29. KL Group, Getting Started with Jprobe Profiler, Version 1.1, RefNo. GETSTRT-JPP/31-03/98. KL Group, 1998.
30. Martin J., Odell, J., Object-Oriented Methods: A Foundation, 1995.
31. Nikander, P., A Java Beans Component Architecture for cryptographic protocols. Telecommunications software and multimedia, Helsinki University of Technology, 1998.
32. Nikander, P., Parssinen, J., Sahlin, B., Hoglund, K.. A Java based framework for cryptographic protocols. Telecommunications software and multimedia, Helsinki University of Technology, May 1997.
33. Orfali, R., Harkey D., Client/Server Programming with Java and Corba, 2nd edition, John Wiley & SONS, Inc., 1998.
34. Parssinen, J.. Java Protocol Framework. Telecommunications software and multimedia, Helsinki University of Technology, April 1997.

35. Platt, D. S., The Essence of OLE with Activex, Prentice-Hall, Inc., 1997.
36. Rational Software Corporation, "Unified Modeling Language: Notation Guide, Version 1.1," Rational Software Corporation, Santa Clara, CA, September 1997.
37. RFC1825. Atkinson, R. Security architecture for the internet protocol. 1995.
38. RFC768. Postel, J.. User Datagram Protocol. Menlo Park, California: SRI International, 1980.
39. RFC791. Postel, J.. Internet Protocol. Menlo Park, California: SRI International, 1981.
40. RFC792. Postel, J.. Internet Control Message Protocol. Menlo Park, California: SRI International, 1981.
41. RFC793. Postel, J.. Transmission Control Protocol. Menlo Park, California: SRI International, 1981.
42. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., Object-Oriented Modeling and Design, Englewood Cliffs, NJ, Prentice-Hall, 1991.
43. Smith, Connie U., Performance Engineering of Software Systems, Addison-Wesley Publishing Company, 1990.
44. Stallings, W., Data and Computer Communications. Macmillan Publishing Company, 1994.

45. Szyperski C., and Pfister C. (1997) Workshop on Component-Oriented Programming, Summary. In Muhlhauser M. Special Issues in Object-Oriented Programming - ECOOP96 Workshop Reader. dpunkt Verlag, Heidelberg.
46. The ATM Forum, Private Network-Network Interface Specification Version 1.0 (PNNI 1.0), pn timer-0055.000, March 1996.
47. Williams, L., and Smith, C., Performance Evaluation of Software Architectures, WOSP98, Santa Fe, 1998.
48. Zweig, J. M. An Object-Oriented Framework for Implementing Network Protocols. University of Illinois at Urbana-Champaign, 1991.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved